DYNAMIC ANALYSIS OF EXECUTION: POSSIBILITIES, TECHNIQUES AND PROBLEMS

Birol Omer Aygun

Carnegie-Mellon University

Prepared for:

Defense Advanced Research Projects Agency
Air Force Office of Scientific Research

September 1973

UNC
SECURITY CLASS THIS PAGE (When Data Entered) I AD-784 882

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1 REPORT NUMBER: AFOSR - TR - 74 - 1430	2 GOVT ACCESSION NO
3 RECIPIENT'S CATALOG NUMBER	
4 TITLE (and Subtitle): DYNAMIC ANALYSIS OF EXECUTION: POSSIBILITIES, TECHNIQUES AND PROBLEMS	5 TYPE OF REPORT & PERIOD COVERED: Interim
	6 PERFORMING ORG. REPORT NUMBER
7 AUTHOR(s): Birol Omer Aygun	8 CONTRACT OR GRANT NUMBER(s): F44620-70-C-0107
9 PERFORMING ORGANIZATION NAME AND ADDRESS: Carnegie-Mellon University, Department of Computer Science, Pittsburgh, PA 15213	10 PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61101D, A0827
11 CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Blvd, Arlington, VA 22209	12 REPORT DATE: September, 1973
	13 NUMBER OF PAGES: 193
14 MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Air Force Office of Scientific Research, 1400 Wilson Blvd, Arlington, VA 22209	15 SECURITY CLASS. (of this report): UNCLASSIFIED
	15a DECLASSIFICATION/DOWNGRADING SCHEDULE

16 DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17 DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18 SUPPLEMENTARY NOTES

19 KEY WORDS (Continue on reverse side if necessary and identify by block number)

20 ABSTRACT (Continue on reverse side if necessary and identify by block number)

The problem of designing computer systems which are far more helpful to the user than current systems in dynamically analyzing program behavior is studied. The functional requirements which such a facility must meet are outlined. The fundamental objective is to permit the user to analyze a program in terms of a user-defined level of abstraction suitable to his particular analysis. A prototype implementation which meets most of the requirements is described. The implications of such a facility for machine architecture to reduce execution overhead are explored.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

I-A

DYNAMIC ANALYSIS OF EXECUTION:

Possibilities, Techniques and Problems

by

Birol Ömer Aygün

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213
September, 1973

Submitted to Carnegie-Mellon University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107) and is monitored by the Air Force Office of Scientific Research.

CARNEGIE-MELLON UNIVERSITY
COMPUTER SCIENCE DEPARTMENT

THESIS ABSTRACT

DYNAMIC ANALYSIS OF EXECUTION: Possibilities, Techniques and Problems

by

Birol Ömer Aygün

The problem of designing computing systems which are far more helpful to the user in the analysis of a program's behaviour at run-time, than current systems is studied.

By considering four application areas, namely debugging, flow analysis, performance measurement and storage reference pattern analysis, a list of specifications for a "general-purpose execution analysis facility" (GPEAF) are drawn.

A prototype facility, called DAME (Dynamic Analysis and Modelling Environment), implemented on the PDP-10 for studying the behaviour of PDP-11 programs, is described. DAME contains a PDP-11/20 simulator and a programmable analysis facility. It is shown that DAME satisfies most of the above requirements.

Significant aspects of DAME are: (i) Access to the state of the PDP-11 at memory and register cycle level, (ii) A flexible hook mechanism which permits arbitrary analysis computations at many points in the instruction cycle, (iii) A node mechanism which permits the user to define over his program a "level of abstraction" suitable for the desired analysis, (iv) A comprehensive instruction set for analysis procedures.

The node mechanism, perhaps the most novel feature of DAME, enables the user to define at run-time a set of "nodes" in his program, in terms of which the execution will be monitored. A node is a portion of code, viewed as a "black box", having unique entry and exit points. During execution, DAME constructs a set of the inputs and the outputs of each occurrence of each node. The node mechanism permits backtracking to any point in the execution history, and control and data flow analysis at node level.

Five detailed examples of the application of DAME to analyses, difficult or impossible with other systems, are given. Example 1 illustrates the input/output sets of nodes and accessing the previous values of an address. The PDP-11 program used in

Example 1 through 4 is a recursive Quicksort program. Example 2 illustrates the determination of the transition frequency between nodes and Example 3 analyzes the parallellism in the Quicksort routine at the recursive call level as examples of control flow analysis. Example 4 illustrates analysis of data flow between two consecutive nodes by comparing the output-set of the first with the input-set of the second. Example 5 illustrates a procedure for the analysis of the instruction mix and addressing modes used by PDP-11 programs.

The present performance of DAME is poor due to simulation at memory cycle level and checking for monitor actions at every memory and register access. It runs 1000 to 2000 times slower than a PDP-11/20 when input/output sets are not used, and 4000 to 5000 times slower when they are. Measurements indicate that respective speed ratios of 300 and 2500 for the above cases are achievable without major re-design.

In designing analysis facilities for ALGOL-like languages, while the main features of DAME are still applicable, other complexities arise (e.g. scopes of variables, recursion, selecting a "unit of execution"). These problems and some approaches to their solution are illustrated for a subset of the BLISS language.

To be economically feasible, systems such as DAME will require assistance from hardware. Microprogrammed implementations of hook and node mechanisms, involving tag bits, associative table searches and monitoring for special bit patterns to detect hooks, are studied. Key problems are seen to be access to the complete state of the monitored machine, interference due to resource sharing with the analysis facility and scarcity of microstorage.

FOREWORD

The research which resulted in this dissertation may be viewed as a journey through a neglected area in computer science. While most areas in computer science are in very primitive stages of development, the area of dynamic analysis of program behaviour is certainly one of the most neglected and potentially most beneficial for both programmers and users of computers.

A look at the Table of Contents will show the reader the many dimensions of this problem which had not received a systematic examination up to now. Thus, the dissertation itself may be regarded as a map of this heretofore neglected region, identifying its major components and the relations among them. Inevitably, all components have not been studied in the same degree of detail. However, hopefully, enough detail and insight have been provided for the crucial parts to give a head-start to the worker interested in designing such a system.

In retrospect, I would like to acknowledge with gratitude the contributions of many individuals in various stages of the research and thesis preparation. Professor David Parnas, a member of CMU Computer Science faculty for most of the period over which this research took place, provided valuable advice during the formative stages of the research and during an earlier implementation of a monitoring facility. Professor William Wulf provided both general guidance and specific technical contributions to the architecture and participated in the evaluation of that facility. He also took part, with Professors Jack Mc Credie, Sam Fuller and Mary Shaw, in the evaluation of the thesis proposal and the progress of the research. In particular, he provided a key idea in Chapter 7, which deals with execution analysis facilities for high-level languages.

Special thanks are due the members of my thesis committee, Professors Jack Mc Credie (Chairman), Victor Lesser, Sam Fuller, Raj Reddy and Andrew Wong, for generously contributing their time to the reading and discussion of the dissertation, for numerous corrections and suggestions for improvements, and re-reading the revision.

I am particularly grateful to Prof. Jack Mc Credie for the continuous dialog, guidance and support he provided in both technical and administrative matters related to the research and the thesis. While the members of the thesis committee and others have contributed much to the technical soundness and to the form of the presentation of the thesis, I bear the sole responsibility for any errors and any technical or editorial deficiencies.

As with most projects involving a complex computer program utilizing tools developed by others, thanks are due to a number of fellow workers. These include Amund Lunde, now with the University of Oslo in Oslo, Norway, Roy Levin, Mady Bauer, Richard Johnsson, Joe Newcomer, Chuck Weinstock, Mario Barbacci, Dave Wile, Jerry Apperson and numerous others upon whom I have called for help on many occasions.

Thanks are due the CMU Computer Science Department for providing an atmosphere highly conducive to research, as well as access to an excellent computer facility and financial support for four years.

Finally, I must express my profound appreciation for the encouragement and support of my family, without which I may not have been able to persevere through the long period of undergraduate and graduate study and research. My deepest gratitue is to my wife, Güzin, who has not only typed, edited and re-typed the entire thesis several times over while holding a full-time job, but also sacrificed many social activities and diversions in order to provide the home atmosphere and peace of mind I needed to carry on my research. Without her unwavering help and energy, I do not know when this thesis would have been completed.

CONTENTS

CHAPTER 1

INTRODUCTION AND MOTIVATION

1.1 Execution Analysis Defined

As the impact of computer technology pervades essentially every aspect of contemporary civilization and as we relegate more and more responsibilities to the computer, it is reasonable to expect that programmers, analysts and users of programs will need more and more powerful tools to analyze the behaviour of programs they are concerned with. The kinds of analyses one can immediately think of include, but are not limited to, debugging, performance measurement, validation and certification. As the complexity of programs grows far beyond the ability of any one individual or a small group of individuals to completely understand and predict their behaviour at any level which is of interest (as it already is today with most large programming systems), the need for better tools to answer questions about the workings and the behaviour of programs grows proportionately. I shall use the term "execution analysis" to include any inquiry into the behaviour of a program, normally in a specific class of environments. I shall leave the word "program" undefined, relying on its intuitive meaning, except to require that the analyst be able to identify what is to be considered as a part of a program and what is not. I shall further concentrate on the execution of programs on computers similar in basic architecture to those in most common use today, i.e. in which each processor has a single instruction stream and addresses a linear primary memory, at least as seen by the programmer. By execution analysis, then, more specifically, I shall mean inquiries into the machine states, and the relationships among machine states, which are evoked by a particular set of executions of a particular program on such a machine.

1.2 Objectives of Thesis

The main objective of the thesis is to report on a research project into the design of environments which would facilitate a very broad range of execution analyses. Of particular interest are:

(i) representation of execution history information in a manner which facilitates the introduction of high-level constructs for describing and capturing diverse aspects of program behaviour,

(ii) a particular set of such constructs which provides a kernel for a large class of analyses,

(iii) extendability of the provided set,

(iv) a general-purpose programming facility for sensing (arbitrarily complex) conditions and taking associated actions at any point during the execution the program under analysis.

In addition to these, I shall consider, in less detail, the extension of the presented ideas to high-level languages and the architectural implications for machine design arising from them.

1.3 Major Application Areas

Although questions related to execution analysis pervade every area of computer science and technology, for the purposes of concreteness, I shall select and examine in detail several of the more prominent ones. The objective of this examination will be to arrive at a set of functional requirements for an analysis facility which will substantially facilitate such analyses.

1.3.1 Debugging

I do not wish to dwell unnecessarily on the fundamental importance of the debugging problem and its magnitude. Let a quote by J. T. Schwartz from a recent symposium on debugging systems [RU 1971] suffice: "Normally, at the beginning of the debugging process, even a programmer with some past experience can never believe how bad things are really going to be before the end." Schwartz also gives a thoughtful exposition of the classes of bugs plauging the field. Here, I shall follow roughly his approach to present a taxonomy of debugging problems.

Let me first delineate the field of "bugs" from all other forms of error in programs: specifically excluded are (i) syntax errors, (ii) errors due to total incompetence on the part of the programmer; that is, if a program is so far off in design and implementation from performing its intended function that it would have to be completely, or almost completely, rewritten to make it work, I shall not consider such a circumstance a "bug". Thus, I shall consider an error a bug if and only if it can be corrected by changing a small part of the total program, although it may well have taken quite a long time to isolate it. This limitation I place on the kinds of errors I shall call bugs is necessary, because otherwise one runs into the strongly unsolvable problem of proving (or disproving)

the equivalence of algorithms in general.

The first class of bugs I shall mention (following Schwartz) are those due to losing count of things: e.g. exceeding array bounds, one too many or one too few iterations in a loop. Another common class is due to omission of initialization of variables, "manifest in situations in which things fail to have either the initial or terminal value which a programmer expects". These are examples of bugs which usually manifest themselves in the early stages. In later stages, "situational" bugs become apparent in the interfaces between relatively distant parts, i.e. in the assumptions those parts make about each other (following D. Parnas's definition of "interface"). "Semaphore bugs" and timing bugs also generally belong to this class, their existence being due to lack of cooperation among the various parts. Also, in this large class, is the set of bugs due to not reading the language or system reference manual properly or to errors in such publications, especially with respect to services provided by the system, e.g. meaning of a particular bit combination in a device control register or the parameter-passing conventions for a system macro.

Schwartz next considers "various aspects of the habitat of bugs". In languages which permit the use of pointers, in particular assembly and higher-level implementation languages, "one often transfers off to nowhere or begins writing into some strange place". (This author has received countless "illegal memory reference" messages from the operating system during the development of this project and wished there were a debugging system with the facilities desceibed in this report, although he did have access to some of the better debugging facilities provided by current systems).

In his thesis entitled "The Debugging of Computer Programs", R. Stockton Gaines provides a more structured taxonomy of bugs, which is summarized below:

"1- Point of origin in the programming process: that is, whether the bug arose in the formulation of the program, or during its implementation...

2- Whether in data definition or data manipulation...

3- Control and Computation bugs...

4- Bugs resulting from lack of knowledge or misunderstanding of features of the operating environment...

5- Fatal and non-fatal bugs...

6- The point at which the bug may be detected. Some may be detected automatically, (that is, in a purely mechanical fashion by checks provided in the compiler or in the generated code or operating system), while others can only be found by intelligent activity on the part of the programmer."

The most characteristic feature of the debugging activity is the search for the cause of an unexpected program behaviour which has just been observed. If one attempted to diagnose the observed anomaly through the examination of a voluminous set of unstructured execution trace data, one would often have an insurmountable task. Hence, the aim of debugging tools is to narrow down the amount of data to be looked at "with the intent of locating an operation transforming reasonable arguments into an unreasonable result" (Schwartz). The risk involved in reducing the amount of data collected, of course, is the possibility of leaving out some important information about the program behaviour which could lead to the isolation of the bug. Hence, given a debugging system which the user can direct to collect certain kinds of information, one measure of the power of the debugging system is the degree of precision with which the user can specify what kinds of data he wants collected. Other measures are the ease with which the user can state his specification and the time between the iterations of a debugging step.

Normally, the first aim of the programmer in the debugging process is to put bounds on the portions of execution history involving improper program behaviour. This requires an ability to move back and forth easily in the execution history, to observe the data flow as a function of control flow and vice versa.

This brings us to the area of flow analysis, which has applications in many areas beside debugging.

1.3.2 Flow Analysis

By "flow analysis" of a program P, I shall mean inquiries into the relations between sequences of machine states which arise during a set E(P) of executions of P. The set E may be small, or large enough to be considered infinite. For example, consider a sorting program S(a,b) whose parameters a and b are the starting and ending addresses of a vector of integers to be sorted. Then the set of all executions of S(a,b) for all a=0,...,[n/2]- and b=[n/2]+,...,n where n is the number of core locations in user address space, is essentially infinite. (By [k] and [k]+, I denote the "floor" and "ceiling" of k respectively.)

A typical problem in flow analysis is determining the set of all successors of every node in the program and the associated transition probabilities. The result is normally expressed as an mxm matrix M where M(i,j) is the probability of node j being the next node if the current node is i. This problem can be extended to the determination of the set Q(i,k) of all k-node sequences following node i. This extended problem may be regarded as the determination of "path-traversal probabilities" and requires somewhat more elaborate machinery than the determination of single step transition probabilities. This class of models of control flow is generally called "Markovian models" and, under certain assumptions of independence of past program behaviour, yield probabilistic information about patterns of behaviour. Coupled with estimates of CPU usage at each node, these models can give resource usage estimates over arbitrarily long paths.

Another example of flow analysis is that used in predicting the degree of parallellism which can be obtained. In this case, one tries to determine which parts of a program could be run independently of which other parts and at which point in the execution. In general, parts which do not communicate at all can be run in parallel. The problem often becomes one of decomposing the program into parts which either do not communicate at all or whose communication permits synchronization of their execution while still maintaining a substantial amount of overlapped, parallel execution. The determination of "communication" between two parts of a program can be very difficult. For example, whether or not part A "tells anything" to part B may be input-dependent in a complex way. I shall call this problem the "data flow" problem. Data flow, together with control flow, constitutes the essence of a program's logical behaviour. I shall deal with this problem at length in the rest of the thesis.

1.3.3 Performance Measurement

Two types of performance measurement have already been discussed under Flow Analysis. Another, more general type of performance measurement problem is the timing of arbitrary paths through a program. We may wish, for example, to define, several paths through a program; start timing when one of these paths is entered and stop timing when the execution deviates from it. We may decide to keep or discard measurements of partial traversals of a path. We may further wish to start the measurement activity only after certain events have happened; e.g. after a certain routine has been called a fixed number of times. This might be desirable, for example, in evaluating the time spent in a space-management routine only when it is called in the "main-loop" portion of a program, after the initial allocation

of space has been made. Thus we need control over which paths are to be measured and under what conditions they are be measured.

In timing a program P running in a time-sharing environment, the other programs running concurrently with P have a certain amount of effect on the measurements on P. As an example, in some time-sharing systems, the overhead for handling an interrupt is charged to the program which was running when the interrupt came in, which is not necessarily the one to which the interrupt belongs. It should possible in a general-purpose execution analysis facility to measure accurately the time taken by a program, as well as usage of other system resources, e.g. main storage. This brings us to a class of analyses which are traditionally done by post-mortem processing of a tape file containing the sequence of addresses generated by means of a hardware probe during the execution of the program whose behaviour is under analysis. These analyses, which are especially important in paged systems, are called "Storage Reference Pattern Analyses" and are discussed further in the next sub-section.

1.3.4 Storage Reference Pattern Analysis

For the analysis of programs from a paging point of view, one can identify several major variables: the hardware (in particular page size and the paging store), the operating system (in particular, its paging policies), the system load and the particular program we are analyzing (in particular page-reference patterns). In theory, it is possible to hold one or more of these variables constant and vary the others. However, in practice, one most often has to hold at least the first two constant, live with uncontrolled variations in the third and try to improve the fourth.

It is always beneficial to analyze the static reference pattern of a program from the program text. One can achieve a certain amount, perhaps a great deal, of improvement this way. However, in general, the storage reference pattern is a function of the inputs. Therefore, one needs to gather dynamic information on the page-reference behaviour on various parts of one's program. There is no easy way to get this information at present. There are hardware devices for measuring all the page references in a computer system over a specific interval of time. Not only are these devices hard to get and make routine, practical use of, they are also not dynamically controlled; so, it is not possible to monitor only a specific part of a program. Clearly, a more flexible tool for obtaining and analyzing this data is needed.

In this sub-section, I have discussed four areas (namely debugging, flow analysis, performance measurement and storage reference pattern analysis) for the application of execution analysis techniques described in this thesis. In Chapter 4, I shall take specific problems from these four areas and illustrate the usage of the prototype software facility DAME in solving them. In the next sub-section, I shall survey the state-of-the-art in execution monitoring facilities.

1.4 State of the Art in Dynamic Execution Analysis Tools

The first remark one can make regarding the state of the art in this area is that it is almost non-existent outside the sub-area of debugging tools. A further indication of the state of the art in execution analysis is the fact that all the on-line debugging tools which have come to this author's attention could be used for various other types of execution analysis but they hardly ever are; in fact, with minor extensions they could be make into quite useful execution analysis tools. The software technology has failed to properly utilize even its existing tools in execution analysis. This could be attributed to their being labelled "debugging aids" as well as to not requiring meaningful execution analysis data from programmers in the industry as well as in the universities. At any rate, the following list is representative of the types of machine language debugging facilities found in major systems: (e.g. see [BO 68])

1- Setting and removing breakpoints at arbitrary points in a program,

2- Computing arbitrary functions of the state of the user-addressable core at a breakpoint,

3- Referencing core symbolically,

4- Transferring control to an arbitrary core location at a breakpoint,

5- Calling other debugging procedures,

6- Modifying contents of core,

7- Defining symbols private to the debugging system and using them as normal identifiers in debugging procedures,

8- Specifying automatic collection of the values of specific locations,

9- Directing dumps and traces to user-specified devices.

As mentioned earlier, these abilities form a basis upon which more useful analysis systems could be built. However, the use of these facilities appears to have remained largely in debugging.

I would now like to mention several systems whose facilities have gone in somewhat different directions. In [ST 65] T. G. Stockham describes a graphical debugging system which has the ability to interact with the user during the course of the execution in terms of the flowchart of the program, a significant advance in man/machine communication. D. U. Wilde wrote a program which statically charted the data and control flow in IBM 7090 programs and attempted to construct functional expressions relating the interacting variables [WI 67]. Its main limitation was limiting itself to static analysis. More recently, the Symbol computer developed by Fairchild Corporation permits the user to specify a routine to be activated when user-specified locations are accessed. This ability forms the basis for significant advance in execution monitoring and analysis capabilities. A similar feature was described for an Algol-like language by J. Mc Neley [MCN 68]. R. Balzer's EXDAMS system [BA 67], though it never became operational, made a attempt to provide high-level facilities for obtaining traces, extracting information from it and displaying it flexibly on a display tube and running the program forwards and backwards. The work by T. Cheatham and his associates at Harvard toward a software laboratory included components for monitoring the value of user specified predicates while a program runs on a real machine or a software interpreter of machine language.

Various machine-simulator based debugging systems have been built and reported. The MIMIC system described by R. Supnik, the AIDS system of R. Grishman and the HELPER system described by H. Kulsrud are some good examples of such systems. (See [RU 71] for reports on these systems.) The main limitations of these systems are: (i) The very limited amount and types of computational power which they appear to have been designed to provide, (ii) They offer no higher-level unit of programming than individual instructions, e.g. the effect on the machine state of five consecutive instructions storing into the same location would be recorded as five seperate entries and there is no way to change this; thus they very quickly run into main storage problems. The ability on the part of the user to define a global structure over his program containing elements of arbitrary size and to be able to capture arbitrary information about data and control flow between these elements would greatly facilitate the analysis of interesting questions about the program's run-time behaviour.

Another class of systems which is relevant to this topic is the so-called "virtual machines". (For a set of papers on virtual machines, see [ACM 73]. However, in virtual machines reported so far, no analysis facilities or features for user control of the computation significantly better than the breakpoint-oriented debugging facilities (such as PCS [BO 68]) of interactive systems have been described.

CHAPTER 2

FUNCTIONAL REQUIREMENTS FOR A GENERAL-PURPOSE EXECUTION ANALYSIS FACILITY

In this chapter, I would like to review and classify the functional capabilities required to accomplish the classes of tasks outlined in Chapter 1 and to arrive at a set of functional specifications for what can truly be called a "general-purpose execution analysis facility" (GPEAF). By "functional specification" I mean that only "what" is wanted is to be specified, leaving the method of implementation open. Having made this statement, let me violate it just once, in order to give a lot more concrete context to what is to follow: If we view the kinds of analysis tasks which have been mentioned as points in a space of infinitely many, continuous dimensions, then the set of functional capabilities of a GPEAF can be viewed as a set of primitive operators and data structures which, when used in composition, juxtaposition and iteration in normal programming style, permit one to proceed easily to most points in that space. This statement will serve as a qualitative specification of the overall function of a GPEAF.

2.1 Debugging

Let us first consider the kinds of questions that arise most commonly in the debugging process. Recalling the types of classifications of bugs given in sub-section 1.3.1, probably the most promising breakdown is into "Control" bugs and "Computation" bugs. I do not wish to imply that I believe these two classes are independent, rather simply that in thinking over all the hours (days, years) I spent debugging programs, it seems that a great deal of those bugs could be comfortably placed into one of these two classes.

2.1.1 Control Bugs

Control bugs most often appear locally, in the form of errors in conditional branch statements or in the number of iterations in a loop. While the actual nature of each error in control flow is, of course, specific to the particular program, the kind of action one would like to take to diagnose such a bug, would be to be able to say something like: "If I have just done X and the machine state is Y, take diagnostic action D_i if the next state is Z_i ". Here, X_i may take the form of a list of calls on subroutines possibly

with specific values or with a specific relation over the values, instruction addresses or other partial specifications of the instruction with specific operand addresses or values. Example: "After S1 has called S2 twice with parameters A, B and C such that A>B>C, followed by n calls on S3 by S2, followed by two MOVEs to location k, do...". Note that the word "follow" may be taken in its strict sense, i.e, "immediately follow", or simply to mean "come sometime after". Further, it is unclear whether the diagnostic action is to be taken only on the first occurrence of the specified condition or on all its occurrences. It should be possible to formulate all of these alternative interpretations. Also note that the specifications over the actual parameters of a subroutine call require that the analysis facility be able to determine, or, failing that, be told by the user, the locations of the parameters. The machine state specifications Y_i and Z_i are partial predicates involving, possibly complex, functions over the state of the memory, including any general-purpose or device registers.

The diagnostic action D_i may involve, minimally, suspending the execution and displaying certain elements of core. In addition, we may wish to compute the value of a function and store or display its result, automatically continue execution from the same or a different point, or we may wish to backtrack to an earlier point in the execution history. This last requirement, namely backtracking, involves two parts:

1- The specification of the point B to which we wish to backtrack, and the associated search over execution history,

2- The actual backtrack operation.

Having made a backtrack, one may wish to execute a certain number of instructions and jump forward to an intermediate state, and eventually resume execution from the point where the original backtrack command was issued.

Note that the form of the debugging request given above does not cover predicates involving the time-series of the values of a location, e.g. those of a variable whose value is modified in each iteration of a loop. This leads to the general concept of the time-series of the values of a variable - which appears to be a natural and useful construct for debugging. I shall refer to this as the "value-trace" of a variable. The number of values to be kept should be user-specified.

2.1.2 Computation Bugs

Under this title I shall include errors in "formulas", generally characterized by a sequence of arithmetic operations concluded by an assignment. They are distinguished from Control Bugs by the trivial, localized control flow involved. In this class of bugs, we are concerned with the past and current values of variables as well as the new values to be assigned to them in a particular instruction or set of instructions. Note that some of these values may in fact be addresses of indirect operands. Hence we are interested in all the operands (including intermediate pointers, side-effects such as setting of the condition code and automatic incrementation or decrementation of registers) involved in an instruction as well as their relation to the instruction. For example, we want to know not only that instruction I fetches something from address A but also whether A is the eventual source operand, a pointer to the eventual source operand, the eventual destination operand, or a pointer to the eventual destination operand, etc. Hence, for any particular machine, there needs to be a characterization of every operand involved in every type of instruction in its instruction set and a corresponding mechanism in the analysis facility which permits one to refer to each of those operands through its relation to the instruction. Facilities such as this permit one, for example, to say: (i) "If I ever multiply (any number) by a negative number and store the result into X, let me know and stop"; or (ii) "If the truncation error involved in an integer division, defined as abs(1-((destination operand*result)/source operand)), ever exceeds 5%, do ...". Let us note an ambiguity in the former request(i); often the result of a computation, such as the multiplication in this case, is stored temporarily in a different location than its eventual destination. Later, perhaps after several instructions, it is moved to its eventual destination. This is particularly true about machines which do not have memory-to-memory operations. In such machines, the high-speed registers are used to hold temporary results very often. In many cases a temporary result may remain in a register over several instructions. In such a case, how should the "store into X" be interpreted, as an "immediate store", a "store within a fixed number of instructions" or an "eventual store" meaning a store sometime before the computed value is modified? The answer to this question is the same as the answer to earlier questions about interpretations of requests: namely, that it does not matter; every interpretation should be able to be formulated within the analysis facility.

I would now like to mention a construct and an associated notation first used (to the best of my knowledge) by C.A.R.Hoare

The below quote is from E. W. Dijkstra's "A Short Introduction to the Art of Programming":

'Let P, P1, P2, ... stand for predicates stating a relation between values of variables. Let S, S1, S2, ... stand for pieces of program text, in general affecting values of variables, i.e. changing the current state. Let B, B1, B2, ... stand for either predicates stating a relation between values of variables or for pieces of program text evaluating such a predicate, i.e. delivering one of the values true or false without further affecting values of variables, i.e. without changing the current state. Then

P1[S]P2

means: "The truth of P1 immediately prior to the execution of S implies the truth of P2 immediately after that execution of S"...'. Dijkstra then goes on to state some theorems relating Pi's, Bi's and S. The relation P1[S]P2 given above seems to be another natural and useful construct for debugging purposes: it is a succinct formulation of a question about the effect of a piece of code S on any part of the machine state. It is clearly necessary that such relations for arbitrary P1, S and P2 be testable easily within the analysis facility.

Let us now summarize the capabilities implied by the examples of debugging activities so far:

D1- Determining the path of control flow down to the instruction level,

D2- Determining the type of instruction being executed,

D3- Following arbitrarily complex pointer chains in core,

D4- Determining the addresses and values (old and new) of all the operands (explicit and implicit) of an instruction as well as their relation to the instruction,

D5- Keeping an arbitrary number of previous values of any address, in an easily accessible form,

D6- Computing arbitrary functions over the current machine state,

D7- Searching execution history (backwards and forwards) for a state satisfying a user-specified predicate,

D8- Efficient restoration of a state found in such a search,

D9- Stopping and starting execution,

D10- Performing any sequence of the operations D1 through D9 at any and each of: operand fetch, operand store, instruction fetch and instruction completion times.

While it is rather imprecise to talk about the "completeness" of a debugging system (or of a system with respect to debugging), one can get a certain amount of reassurance of the sufficiency of these requirements for a debugging facility by convincing oneself that they offer a great deal of help in isolating all the classes of bugs mentioned in Chapter 1, sub-section 1.3.1.

2.2 Flow Analysis

As stated in Chapter 1, by the "flow analysis" of a program P, I shall mean inquiries into the relations between sequences of machine states which arise during a set, E(P), of executions of P. It is helpful to think of the program counter as a distinct entity from the rest of the machine state. In machines having a built-in stack, it may also be useful to think of the stack pointer as a third distinct entity, especially in high-level languages which do not permit explicit access by the user to the elements in the stack. I shall not do so here, since I shall be mainly concerned with machine language programs where everything is essentially global.

Thus, thinking of the program counter (PC) as a seperate entity from the rest of the machine state (which I shall call "memory", M), we can identify two types of flow: Control Flow and Data Flow, where the former refers to the sequence of values assumed by the PC and the latter to the sequence of states of M. I would like to emphasize the word "sequence" in the last sentence. The word "flow" implies a sequence of changes to one phenomenon relative to another. Hence, for example, we might ask: "Starting from a particular state of M and PC, what is the kth value of PC?". Or similarly, "Starting from a particular state of M and PC, what is the kth state of M?". Or, "Given that M=M1, when PC=P1, what is M when PC=P2?".

2.2.1 Control Flow

Normally, it suffices to consider only changes from sequential flow in order to be able to reconstruct the entire history of control flow. One must be careful, however, to include enough information to indicate when each change occurred. For example, one may include the starting address of the program, followed by pairs (a_i, b_i), $i=i,\ldots,n$, where a_i and b_i are the origin and

the destination of the ith branch instruction, respectively. Alternately, one might let a_i be the starting address of a block of straight-line code and b_i the number of instructions in that block. A GPEAF should have facilities for sensing the fetch of an instruction from a location X, the completion of its execution, reconstructing the last N branches (origin and target), for arbitrary N. It should also be able to execute a user-specified procedure before and after any or every instruction. (This ability was also listed as a requirement under debugging.) It is important, though this can also be implemented by the user himself using the above facility, that when the user gains control before or after an instruction, he be able to determine the address of the previous instruction. E.g. if one can jump to an address A from several locations, it is necessary to be able to determine easily at A where one came from.

2.2.2 Data Flow

If we think of data flow as a sequence of changes to the state of the memory M representing the progress of execution, it becomes clear that in order to be able to analyze it, we must first relate it to control flow. That is, we must be able to determine which changes are associated with which parts of the execution path. In general, many parts of the execution path may result in an identical effect on the state of M. Thus, data flow analysis must be able to determine, where possible, the precise part responsible for any given effect.

Some Fundamental Relations in Data Flow

Let us consider two contiguous parts, A and B, with B temporally following A, in the execution path of a program. Suppose that we would like to know the data flow from A to B. More specifically, we would like to know the set of addresses which are both modified by A and read by B before being modified again, and the values of those addresses upon entry into B (in the absence of any outside interference, this is equivalent to the values of those addresses upon exit from A). Let us define as the input-set, I_A, of A the set consisting of pairs (a_i, v_i) where a_i is the ith unique address from which A reads something before writing into it, and v_i is the value read.

Let us also define as the output-set, O_A, of A the set consisting

of pairs (b_i, u_i) where b_i is the ith address written by A and u_i the contents of b_i upon exit from A.

Then, the data flow, $D_{<AB>}$, from A to B can be characterized simply as:

$$(1) \quad D_{<AB>} = O_A \cap I_B$$

Note that in computing this intersection, it suffices to look at only the address parts of the elements of the two sets, since, due to the temporal adjacency of A and B, equality of addresses will imply equality of contents. However, no harm will result if, in order to maintain the conventional set-theoretic definition of intersection, we require that only those elements which are identical in all respects (which are used to include them in their respective sets in the first place) be included in the intersection. Hence, the conventional definition of intersection in set theory will suffice for the relation (1).

Let us now consider an important step in data flow analysis, namely the compaction of two consecutive parts into one. This step is fundamental to many types of flow analyses; see for example [CO 71]. To do this, we shall need the following additional notations:

$C(a,t)$ = contents of location a at time t,
$T_e(A)$ = time of entry into part A,
$<AB>$ = the part consisting of the temporal juxtaposition of parts A and B.

We can now characterize the input set of <AB> as follows:

$$(2) \quad I_{<AB>} = I_A \cup (I_B - O_A)$$

Here again, it suffices to consider the conventional set-theoretic definition of the union operation, since the equality of the address part of an element in I_A, to that of an element in $(I_B - O_A)$, implies the equality of their value parts. This can be briefly proved as follows:

Proof: Suppose that for some $p=(a_p, v_p)$ and $q=(a_q, v_q)$,

$p \in I_A$ and $q \in (I_B - O_A)$ respectively, $a_p = a_q$ and $v_p \neq v_q$. Now, since A and B are consecutive, $v_q = C(a_q, T_e(B)) = C(a_q, T_x(A))$, i.e. the contents of location a_q are unchanged between the exit from A and entry into B. But since $a_p = a_q$, v_q must also equal $C(a_p, T_x(A))$. Thus, for $v_p \neq v_q$ to hold, this requires that the contents of address a_p be modified during A. But this contradicts our definition $q \in (I_B - O_A)$. Hence no such elements p and q can exist.

Finally, we can characterize the output set of <A,B> as:

(3) $O_{<AB>} = O_B *\cup O_A$

where $*\cup$ denotes an extension of the union operation to one which "favors" the left-hand operand over the right hand one in the sense that, if there is an element (a_L, v_L) in L and (a_R, v_R) in R such that $a_L = a_R$ but $v_L \neq v_R$, then $L *\cup R$ includes (a_L, v_L) in the resulting set.

This operation simply assures that if some address is modified by both A and B, then only the final effect will be recorded in the output set of <AB>.

These three highly intuitive relations form a base upon which many data flow analysis mechanisms can be built.

So far, we have been concerned only with consecutive parts, where we are assured that nobody else will get in between the parts involved in the data flow. But now let us consider the case where the parts, A and B, of the execution path, the data flow between which we wish to explore, are not consecutive. What should the analysis facility be able to tell us about the effects of intervening parts C_i, i=1,...,k, on the data flow from A to B? There are at least two reasonable answers:

1- That the analysis system be able to tell us whether any of the C_i's had any effect on the data flow from A to B or not, or

2- That the analysis system be able to give us a list of the effects, e.g. a list of pairs $((C_1, v_1), (C_2, v_2), \ldots)$ where the first element of the pair indicates the effecting part, and the second element indicates the effect.

A little reflection shows that the first, "yes" or "no" alternative is not satisfactory unless there is some practical way of finding out the information provided by the second alternative. Hence, I shall adopt the latter as the information which the analysis facility must provide.

What Maketh a Part?

All the discussion so far of data flow has been based on the notion of "parts" in a program, which could be treated as units of control flow and which provide the link between control flow and data flow. Now, let us consider what properties a part should have, and how we would recognize one if we saw one.

The main motivation for the introduction of the notion of parts was to break up the entire execution path of a program (with a set of inputs) into more manageable units for gathering, manipulating and interpreting information about data flow. The smallest conceivable unit for this purpose is a memory cycle (between the processor and main memory, or the general registers). For executions involving tens or hundreds of thousands or millions of machine instructions, this would require the recording of about four or five times that many (address, value) pairs. This amount of data is clearly too voluminous to store in main memory. It conceivably could be dumped into secondary storage periodically and searched as needed. However, it would appear that unless one uses very efficient random access methods, this approach would cause intolerable overhead. Further, this fact defeats the goal of ease of use, since normally programmers don't think in terms of the number of memory cycles when doing flow analysis. Hence, a memory cycle appears to be too small a unit for this purpose.

The next level appropriate for consideration as a part is a machine instruction. But we note that the amount of storage required is of the same order of magnitude as above, since we would have to record every address and value involved in the execution of the instruction. A certain amount of saving is possible though if we recognize that most of the operand addresses involved in an instruction are statically fixed. The only ones that are not, are indirectly addressed operands. To take advantage of this, one can develop a technique for constructing a "template" for each instruction in the program, showing the static

operand addresses (all of which may not be apparent in the instruction itself), and relate to it each instance of execution of the instruction and the dynamic operand addresses involved in that instance. Such a technique can at best save less than 50% of the storage required in the preceding alternative.

Let us now move up one more level, to the level of groups of instructions. The first, obvious question is "How big should a group be?". The reason that one tends to raise the issue of size before other issues here, is that so far we have found the two previous alternatives unattractive because of the size of storage required. Once we move to the level of groups of instructions however, we have a great deal of flexibility. For example, we may wish to consider as a part, groups of instructions which correspond to some syntactic programming unit, such as a subroutine or a block. Or we may wish to consider what are usually called "basic blocks" by compiler writers, namely, blocks of instructions having a unique entry point and a unique exit point. (We must remember at this point that, a "part" refers to a part of the execution path, not of the program text; i.e. for groups of instructions, a part refers to a particular execution of one group). Or, even more flexibly, we can let the user define what should be a part. This latter choice has the advantages of controlling the amount of storage required as a function of the length of execution as expected by the user and of having a part correspond to a conceptual step in the solution of the user's problem.

Given all these alternative strategies for defining parts, the criteria for judging the suitability of a particular choice of strategy are:

1- How well does the chosen strategy perform in answering data flow questions?

2- How practical is it to implement?

In Chapter 3, I shall describe one choice and discuss its implementation and performance.

Units of Data Flow

The most elementary unit of data as represented in digital computers is the ubiqutious "bit". On the other hand, by far the largest fraction of processing is done in terms of "words", the size of which varies from computer to computer. Further, a significant amount of processing is done in terms of fractions of words, called "bytes", and a relatively smaller portion in terms of "blocks" of words. In machine languages, "blocks" are rarely used as individual operands in an instruction (a notable

exception being the "transfer block" instruction implemented in certain machines). The "bit" is also very infrequently used as an individual operand. Rather, it is usually employed to express side-effects of certain operations, e.g. the setting of the condition code, the bits in a processor status word and so on. These side-effects are an essential part of the effect of an instruction and hence any analysis facility must represent and give access to them in an adequate way.

The "bytes" come in two flavors (no pun intended): fixed size and variable size, fixed size being the more commonly used. In variable-size-byte machines, such as the PDP-10, one needs both a starting position and a length to characterize a byte whereas with fixed-size machines one needs only the starting position. Bytes also form an important unit of data flow and should be dealt with in full by a GPEAF. For exqmple, in the input and output sets of a part, the location (word address and starting position within the word), size and contents of byte operands should be properly reflected.

The "word" is probably the most appropriate unit for representing the largest fraction of data flow. I do not feel that I need to dwell on the precise definition of a "word", since its meaning for, probably all, commonly used machines today is clear. An interesting class of exceptions to this would be machines, paper or real, for directly executing high-level languages, such as a LISP machine or a SNOBOL machine. In such machines, the selection of the unit of data flow probably ought to be closely related to the primitive data structures of the language (e.g. atoms, lists, strings).

Thus, we can conclude our discussion of appropriate units for representation of data flow by saying:

1- The main criteria for judging the suitability of a proposed set of units are: (a) Is it capable of representing all elements of data flow?, and (b) How efficiently, in terms of storage and interpretation speed, does it represent the great majority of operations?

2- The choice of data flow units has a large impact on the efficiency of the analysis facility and hence its usefulness.

To summarize the functional capabilities required for control flow and data flow analysis tasks, we can list them as follows:

F1- Giving the control to the user (or a user-specified analysis procedure) before or after every instruction, and before or after user-specified instructions,

F2- Dividing the exeucution path into parts as specified by the user and enabling the user to refer to these parts explicitly,

F3- Constructing the input and output sets of parts, as these sets were defined earlier,

F4- Determining the data flow from a part to the following part as per relation (1),

F5- Computing the combined input and output sets of adjacent parts, as per relations (2) and (3),

F6- Determining the effects of intervening parts on the data flow between non-adjacent parts, as discussed earlier,

F7- Enabling the user to access every element of any input set and any output set, and use the address and value parts of the element in computations.

2.3 Performance Measurements

Performance measurements are concerned with relating the resource requirements of a functioning unit to the degree to which it achieves its goals. For example, one might relate the storage and CPU requirement of a compiler to the compactness and efficiency of the object code it produces. A GPEAF should offer the analyst high flexibility in making these measurements.

We can also talk about performance measurements of operating systems. For example, scheduling, storage allocation and paging policies have become the subject of much research and analysis from a performance point of view. An operating system can be "measured" in two ways:

(i) We can measure its component programs just as we measure user programs (i.e. their storage and CPU requirements etc.)

(ii) We can measure the performance of the whole system while it processes a given workload (i.e. in terms of throughput, average response time (for time-sharing systems), paging rate etc.)

I shall refer to the first class of properties as "program performance" and to the second class as "system performance".

Measurement and Modelling of Program Performance

Among the most frequently used measures of program performance are such criteria as :

(1) How long it runs with a certain input,

(2) How it spends its time,

(3) How much main storage it requires.

Measures (1) and (3) are generally provided by the operating system as user accounting data. (2) is usually obtained by using timing packages or explicitly reading the system clock (job-time) within the user program. In either case, the user has to recompile his program to vary the measurements of type(2) he wants to make. Further, in multiprogrammed systems, the CPU and storage charges for running the same program with the same inputs can vary significantly as a function of the other programs running at the same time. I shall not go into the reasons for this. Let it suffice to say that one often can not get a "pure" measure of a program's running time through conventional operating system facilities. Hence it behooves an analysis facility to offer much more help in this area.

Another problem in measuring the performance of existing programs has been what to do about the parts which we do not want to measure but which provide inputs to the parts which we do want to measure. In large programs, program modification, recompilation and re-loading time and effort required for each change one wants to make to deal with this problem, has often made such measurements too cumbersome and time-consuming to undertake casually. It should be possible within an analysis facility to model or "dummy up" the logic and simulate the timing of uninteresting parts of a program and "skip over" them, and execute and measure in detail the interesting parts. (This procedure is quite familiar to those who have done hand-patching of the code produced by a compiler.) Using a GPEAF, one should be able to define a number of paths p_i, i=1,...,k, through a program possibly using the "part" definitions discussed earlier, and measure the time for each complete traversal of each path. This requires that one be able to sense departures from a path at some intermediate point in the path.

Another technique for measuring where a program spends its time is periodic sampling of the program counter. This technique has the drawback that unless the period of sampling is chosen with great care, certain parts of the program may never appear in the samples because of "lock-step" synchronization between the sampling and the pattern of control flow. However this problem can be overcome with a certain amount of analysis. This technique has the advantage of considerably less overhead, compared with other techniques such as timing each subroutine entry

and exit. To permit this technique, an analysis facility must enable the user to schedule "sampling probes" with a dynamically controlled frequency (to overcome the problem mentioned above).

With regard to measuring the storage requirements of a program, since these are strongly tied to the storage reference patterns, I shall discuss those two subjects together in subsection 2.4.

Measurements and Modelling of System Performance

Under this topic I shall consider the measurement of such properties of operating systems as system overhead, CPU utilization, and paging rates (where applicable) as a function of job mix and system design. It might be, reasonably, felt that we are straying afar from our initially stated purpose of the analysis of program behaviour. However, it must be pointed out that the "true behaviour" of an operating system program can not be studied without some experimentation involving the processing of a typical job mix. It is true that studies involving the characteristics of an operating system over several days or weeks of user time probably fall outside the scope of an analysis system of the type envisioned here, although many functions, such as measurement of the average time between interrupts, the storage reference patterns, the average job running time etc., which can be measured by a GPEAF, could be useful in such studies.

There is another, perhaps more interesting, way however, in which a GPEAF ought to be useful in such analyses. This approach involves the modelling of parts of the user workload and of the operating system via analysis system facilities by which one can mimic the logic and/or the resource requirements of these programs, as mentioned earlier under Measurement and Modelling of Program Performance. An example of such a model is a routine in the language of the GPEAF, which simulates a user job which generates an I/O request every K_i, $i=1,..,n$, milliseconds of CPU time, where each K_i may be a random number drawn from a distrubution. Another example might be a model of the page reference pattern of a job. One might take an ensemble of such models of user jobs and model the execution of those jobs under a given operating system, by invoking the facilities of a GPEAF to interface the models with the operating system. One might even model parts of the operating system (such as I/O servicing, scheduling etc.) for purposes of expediency or efficiency.

Let me now summarize the capabilities required for the performance measurement tasks which have been discussed:

P1- Measuring the execution time required by arbitrary, user-defined paths in a program,

P2- Gaining control of execution at specific time intervals or when I/O or other supervisor services are needed by a user program,

P3- Performing arbitrary computations when control is gained,

P4- Simulating the passage of arbitrary lengths of time.

2.4. Storage Reference Analysis

In the design of paged systems, a parameter of interest is the pattern with which user programs will reference pages. This pattern, if it can be estimated for a large fraction of the total workload the system is to handle, can effect the policy for selecting the page to be swapped out to make room for a new page. For example, it may be used to estimate how often the page to be swapped out is likely to be a "dirty" page, i.e. it has been written on since it was brought in, so that it indeed has to be written out.

A popular measure in paged systems is the working-set size of a program, i.e. the number of unique pages addressed by the program.

Measures of paging activity associated with a program are also of interest to the programmer. Clearly, fewer inter-page references means fewer potential page faults. Hence, if a programmer is interested in improving the page referencing pattern of his program, first he must be able to obtain the existing pattern and determine the effects of each change he makes.

There is at present virtually no way for a programmer to get this information directly.

Another type of analysis is that of register usage. It is desirable that as many operands as possible be available in general registers. Hence, it is of interest to identify "dead" periods of registers, which is the period between two successive stores into a register with no intervening fetches from it. During such a period, that register can possibly be used to hold the values of other variables. In fact, whenever a register can be profitably used to hold the values of several different variables (even if this may mean saving and restoring each such value), the efficiency-conscious programmer may want to analyze the patterns of reference to each such register.

There are many other types of analyses which might be called "storage reference analyses" but which I shall not enumerate.

The functional requirements for these kinds of analyses can be summarized as:

S1- Obtaining every address (including registers) generated by the program, when it is generated,

S2- For each generated address, an indication of whether it is an instruction, an operand fetch or a store,

S3- Making arbitrary computations whenever a generated address and the associated indication is obtained.

2.5. Summary of the Functional Requirements

In this section, I would like to summarize the functional capabilities required for the four analysis areas which have been discussed. I have no formal proof that these capabilities form a "complete" set; nor do I pretend to know precisely what the "completeness of an execution analysis facility" may mean. However, certainly it must mean "something more" than the trivial, formal completeness in the sense of being able to compute all computable functions. Below, I give my understanding of what that "something more" is.

We can consider the required capabilities in four classes:

1- What information the analysis facility has access to,

2- At what points in the execution cycle it can gain control,

3- Its instruction set,

4- External appearance and miscellaneous useful features.

2.5.1 Information Requirements of the Analysis System

The Analysis System needs access to at least two address spaces: the address space of the object machine (which shall also be called the "external state of the OM") and its own symbol space. (Some may want to consider the former as an element, e.g. a large array, in the latter.) In particular, every address and register accessible by the object program must be readable and writable by the Analysis Facility. In fact, the access to the object machine address space should be very easy and direct.

If the Analysis System is inefficient in long computations and therefore a need for a linkage to programs written in a compiler level language (such as the one in which the Analysis System may be written) is indicated, then the Analysis System routines should have access to the symbol space of that compiler level language.

It is desirable that the Analysis System have access to the operand addresses and values of the current object instruction (which shall also be called the "internal state of the OM"), without having to decode them itself. Thus, at the end of an instruction cycle, one should be able to say, in effect: "If this is a MOVE instruction and the source operand value is zero, and the destination address is between A and B, then do...".

It is also helpful if a direct indication of the instruction class (double-operand, single-operand, no-operand) is available.

The question of access to the timing of the object machine must also be considered. The Analysis System must be able to read the clock of the object machine or otherwise determine the object machine time easily, at least after each instruction. For some applications, it may be necessary to determine the object machine time after each major (primary memory) cycle or each minor (register transfer) cycle.

While this is not an absolute necessity (as we have shown that we can get by without it in the DAME system), it would be desirable to have access to the user program text and symbol table, so that the user could converse with the system in terms of this own symbols.

It is clear from the foregoing discussions of control flow analysis, that the user, in cooperation with the system, will define a topology or structure over his program for purposes of control flow history. It is also clear from those discussions that empirical data associated with each component of that structure will be generated during the execution of the user program and that this data will be linked to the appropriate parts of the control flow history. Each of these elements of information, i.e. user program structure, control flow history and dynamically-generated empirical data, must also be accessible by the user.

2.5.2 Triggering of Analysis Actions

The user must be able to execute any (meaningful) set of analysis actions after every operand fetch, store, instruction fetch, instruction completion or at specific points in time (i.e. relative to object machine clock). Further, the user must

be able to specify, optionally, address ranges or registers for which the stated action is applicable. In the rest of the thesis, I shall refer to a stated sequence of actions to be activated at one of the above points as a "hook".

2.5.3 The Instruction Set of the Analysis Facility

The instruction set of the Analysis System should contain two classes of instructions:

1- A complement of instructions similar to those of conventional programming languages : these will be used to perform assignment, arithmetic and logical operations, conditional execution, looping, subroutine call with parameters and I/O. In fact, this subset of the instruction set should be a programming language which is "complete in a practical sense". All the computations, such as those encountered in performance analysis or flow analysis, can be potentially done in this subset of the language.

However, as mentioned earlier, in the case that the Analysis System instruction set turns out to be unsuitable for long computations, there should be an escape mechanism through which one can execute subroutines which are written in a more suitable language (possibly the one in which the Analysis System itself is written). If that language has a syntactic construct, similar to a "function" in some languages, which returns a value, then it should be possible to assign the value returned by such a construct to a symbol in the symbol space of the Analysis System.

2- A complement of instructions particularly useful in monitoring and execution analysis. These should include the following operations:

(i) inserting, deleting, enabling or disabling hooks statically and dynamically,

(ii) defining "parts" in the execution path whose input and output sets (discussed earlier under Data Flow, in sub-section 2.2.2) are to be determined automatically and made accessible to the user,

(iii) Searching the input and output sets of previous parts for one which satisfies a user-specified predicate (better yet, making each set available to the user in some systematic manner, e.g. reverse chronological order, letting the user perform arbitrary computations using the elements, i.e. <address, value> pairs, in the set, and tell the system whether he wants to continue the search or not),

(iv) Displaying input/output sets in an appropriate format (i.e. indicating relations between addresses and values, and the "byte" position and size where applicable),

(v) Backtracking to the beginning or end of a part found in a search or specified explicitly by the user,

(vi) Moving further back or moving forward following the execution of some instructions from a "backtracked" position,

(vii) Resuming execution from the point where the backtrack instruction was issued.

2.5.4 External Appearance and Miscellaneous Useful Features

Since the main design goal for the Analysis Facility is to facilitate the performance of analyses of program behaviour, clearly the associated command language should be easy to use and have good error-detection features. It must be noted that unreasonable-looking results obtained by some analysis procedure are, in some sense, "doubly hard" to disprove or verify, since one may have to re-examine both the analysis procedure and the process under analysis to determine the validity of the obtained result. Further, one frequently has to compose analysis procedures in a short period of time, often in an interactive, spontaneous fashion, a condition which increases the probability of making errors.

All of these conditions point to the requirement that the language of the Analysis Facility be simple and terse in syntax, encourage structured programming, and not rely very heavily on remembering many keywords. This last requirement is probably the most difficult to achieve due to the variety of specialized functions which have to be performed in collecting and searching execution history data. Further, the objectives of a powerful language and simplicity of syntax conflict with the objective of not relying heavily on remembering many keywords. However it has been shown that very good compromises can be reached; witness APL and LISP.

These conditions also mean that often the same commands with minor modifications will be entered repeatedly, increasing the possibility of making typing errors each time they are entered. Hence, the analysis facility should possess a "library" capability where frequently used command sequences can be stored and called when needed. Also, a good editing facility for editing both "on-line", i.e. loaded command sequences, and "off-line", i.e. text files, is extremely useful.

In referencing object machine instructions, e.g. tracing them as they are executed, or displaying a block of instructions, the analysis facility should be able to deal with symbolic forms as well as numerical. This ability is available in most on-line debugging systems today.

It must be noted here that the mental picture, on which these functional requirements are based, is that of an interactive, on-line analysis facility. For batch systems, some of the requirements become more severe, and some less. For example, in an on-line system, the user may display the values of some variables and base his next action on what he sees. In a batch system, this is not possible. Hence, the user would like to do the next best thing - namely, program the reasoning process he uses into the analysis procedure. (To the extent that this process can be mechanized, this is even preferable to the visual examination by the user.) This means that especially in a batch sytem, anything that the user would like to see displayed in an interactive system should be available "inside the machine" to analysis procedures. On the other hand, the "terseness" requirement for the analysis language is not as severe in a batch system as in an interactive system.

CHAPTER 3

THE DAME SYSTEM

In this chapter, I shall describe the design of the DAME (Dynamic Analysis and Modelling Environment) system, which has been the major vehicle in my research for implementing, experimenting with and evaluating new ideas. DAME is a facility for studying the logical behaviour and the performance of programs for the PDP-11/20. It consists of a PDP-11/20 simulator and a programmable analysis facility which achieves most of the requirements set forth in the last chapter. The main goal in the design of DAME was to isolate critical problem areas in the design of a general-purpose execution analysis facility (GPEAF), for which solutions had not been developed as yet and to propose solutions to, at least some of, these problems. It was not the intention to develop a finished, tuned-up utility system for general use. Hence, some features for which satisfactory techniques were already known and which would be very desirable in a system for general use, were omitted from DAME since the effort required to implement them did not seem justified in view of their minimal contribution to the research aspect of this project. However, despite such omissions, I have found DAME to be a powerful tool for analyzing program behaviour.

In order to facilitate the reading of this chapter by readers with different objectives, I shall first provide a detailed outline. This outline can also be used as a reference later to quickly locate the section about a particular point, as well as to guide the reader in the first reading to sections of more interest to him.

Outline of Chapter 3

The first topic is the set of data structures underlying the design of DAME. In Section 3.1, I first summarize these structures and then discuss in more detail some of them, namely, the formats of objects and lists as well as certain master lists and symbol tables which play an essential role in the implementation of DAME.

The description of these structures is provided only because they facilitate certain search operations over pre-defined classes of objects. An understanding of them is not required for an overall understanding of DAME.

CHAPTER 3

THE DAME SYSTEM

In this chapter, I shall describe the design of the DAME (Dynamic Analysis and Modelling Environment) system, which has been the major vehicle in my research for implementing, experimenting with and evaluating new ideas. DAME is a facility for studying the logical behaviour and the performance of programs for the PDP-11/20. It consists of a PDP-11/20 simulator and a programmable analysis facility which achieves most of the requirements set forth in the last chapter. The main goal in the design of DAME was to isolate critical problem areas in the design of a general-purpose execution analysis facility (GPEAF), for which solutions had not been developed as yet and to propose solutions to, at least some of, these problems. It was not the intention to develop a finished, tuned-up utility system for general use. Hence, some features for which satisfactory techniques were already known and which would be very desirable in a system for general use, were omitted from DAME since the effort required to implement them did not seem justified in view of their minimal contribution to the research aspect of this project. However, despite such omissions, I have found DAME to be a powerful tool for analyzing program behaviour.

In order to facilitate the reading of this chapter by readers with different objectives, I shall first provide a detailed outline. This outline can also be used as a reference later to quickly locate the section about a particular point, as well as to guide the reader in the first reading to sections of more interest to him.

Outline of Chapter 3

The first topic is the set of data structures underlying the design of DAME. In Section 3.1, I first summarize these structures and then discuss in more detail some of them, namely, the formats of objects and lists as well as certain master lists and symbol tables which play an essential role in the implementation of DAME.

The description of these structures is provided only because they facilitate certain search operations over pre-defined classes of objects. An understanding of them is not required for an overall understanding of DAME.

In Section 3.2, another data structure, the representation of the PDP-11 core, is described. In this connection, I also present the general problem of representing the memory of one computer in another, emphasizing the problems related to the respective memory sizes and word lengths of the two machines.

In Section 3.3, the question of the "time-grain" of simulation is considered. In particular, the costs and benefits of simulation at the memory cycle level and at the instruction level are briefly discussed and compared. (Note: This topic is discussed in more detail in Chapter 5.)

In Section 3.4, the hook mechanism is described. The types of hooks and the points in the PDP-11 instruction cycle at which they may be placed are explained. Whenever a hook is activated, the PDP-11 simulator makes available to the user certain information on the current state of the processor and the Unibus by storing that information in PDP-10 global symbols. In this section, a list of the PDP-10 global symbols used for this purpose is given.

In Section 3.5, the most significant feature of DAME, the Node Mechanism, is described. This mechanism permits a guaranteed backtrack capability to any point in the execution history and an analysis of data flow in terms of user-defined nodes. The user thus has almost complete control over the amount of execution history information collected by the system.

An understanding of the Hook Mechanism (Section 3.4) and of the Node Mechanism (Section 3.5) is essential to the understanding of the rest of the thesis.

In Section 3.6, an outline of the DAME instruction set is given. First the general syntax of DAME instructions is specified. The instruction set is divided into two subsets. The first subset (Section 3.6.1) contains the instructions provided for normal programming operations such as assignment, arithmetic, looping and the like. These instructions are listed without much explanation, except for several instructions which are more uncommon (e.g. a search-list instruction). The latter are explained in detail. The second subset of instructions (Section 3.6.2) consists of those which are specifically designed for monitoring the execution of the -11, collecting data and searching them. These are also explained individually. An understanding of this section shoul be sufficient to follow the detailed illustrations given in the next chapter. However, for those who wish a more detailed and systematic description of the instruction set, a user's manual is provided in Appendix A of the thesis.

In the final section, Section 3.7, some unimplemented ideas

for improving the performance of DAME are discussed. They are immediately implementable, as opposed to future research, ideas.

3.1 The Underlying Data Structures

In DAME, one has access to three address spaces:

1- Objects and list structures, which are the main class of entities that DAME deals with,

2- PDP-11 core, general and device registers,

3- Global PDP-10 symbols used in the simulator.

(Note: These three address spaces are not disjoint; the PDP-11 registers and core are also accessible as PDP-10 global symbols. Some operations which normally operate on objects can also operate on -10 globals.)

In the rest of this section I shall describe the structure and possible attributes of objects, and several global, pre-defined list structures which are crucial to the implementation of DAME. The other two address spaces will be discussed in succeeding sections. The reader who is not concerned with the implementation, can skip to Section 3.2 without loss of continuity.

Most of the information structures generated by DAME during the execution of an -11 program are in the form of lists, as are DAME routines themselves and most of the pre-defined information in the environment. The basic list-processing functions and the PDP-11 simulator are implemented via the BLISS-based general-purpose simulation package POOMAS, developed by Amund Lunde [LU 71].

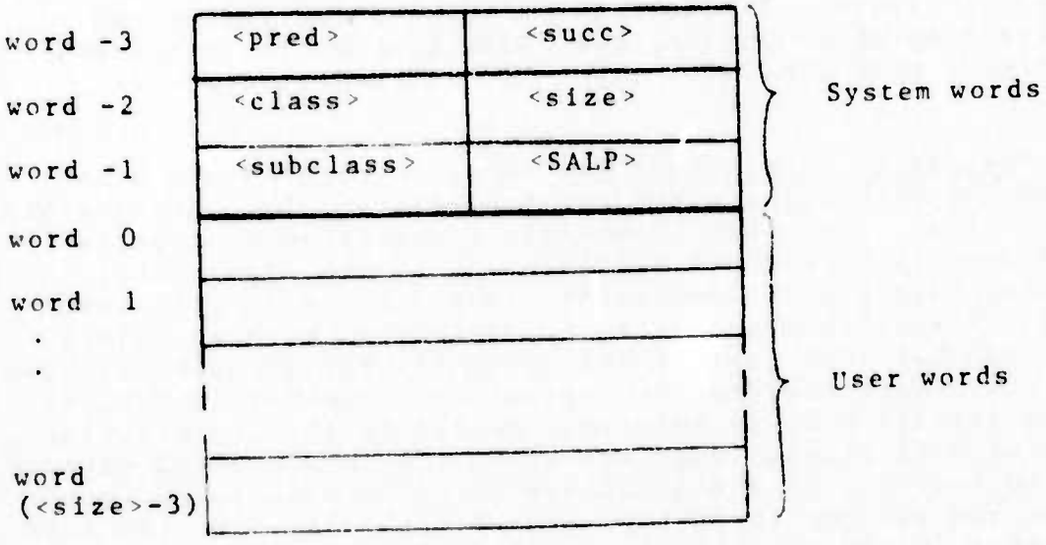
Attributes of DAME Objects

Each DAME object has the following attributes; a "successor", a "predecessor", a "size", a "class", a "subclass" and possibly a list of "secondary attributes". (The first four attributes are provided by POOMAS.) Objects which are not members of any list contain a special code, NONE, as their successor and predecessor attributes.

All of the above attributes of an object, except the secondary attributes, are represented in three "system words" preceding the first "user word" of the object. Objects are addressed by their first user word, called "word 0". The system words are also called "word -1", "word -2" and "word -3". The standard object format is shown in the next figure.

Illustration 3.1

DAME Object Format



<pred> : pointer to predecessor

<succ> : pointer to successor

<SALP> : secondary attribute list pointer

The "class" attribute is used mostly by POOMAS (e.g. to identify list heads, objects representing process pointers and event notices on the simulation event calendar). DAME also uses the class attribute to indicate what is usually called "data types" in programming languages, namely whether some data is normally to be treated as a character constant, character variable, numeric constant, numeric variable, etc.

In addition to the class attribute, DAME objects have a "subclass" attribute. The subclass attribute designates the general function of an object, e.g. DAME instruction subclass, hook subclass, node subclass, input-set subclass, output-set subclass.

The "secondary attributes" are those attributes which may be defined for some objects but not necessarily all. The secondary attributes of an object are themselves represented as objects and are put on that object's Secondary Attribute List (SAL). The SAL also serves as a convenient place for the user to save arbitrary information which is to be associated with an object but which can not be a part of its contents. For example, suppose a user would like to record the contents of some core locations whenever a certain node is entered. He can do that by creating an object of a particular subclass every time the node is entered, copying the contents of the locations he is interested in into the object and putting it on the node-object's SAL. He can later retrieve that information by a special DAME-supplied function by giving the subclass of the secondary-attribute-object.

Subclass Master Lists

In order to provide access to objects via their subclass (i.e. their general function) there is a master list for each subclass, which contains a pointer to every object of that subclass. Thus, for example, it is possible to search the set of all node-objects or hook-objects for one satisfying a particular condition, or to delete all the DAME routines defined so far etc. In particular, there is a subclass called "subclass-master subclass", which contains all these subclass master lists. Most of the objects existing at any given point in time, can be accessed, without knowing their name or address, through these master lists.

Symbol Tables

In addition to the subclass masters, there is a conventional symbol table maintained by DAME, which permits access to the objects by their names. The Symbol Table is also organized as a list and can be searched by the usual list processing functions. Since the user can refer to global PDP-10 symbols, the DDT symbol

table is also present during execution. (A list of some useful symbols is given in the User Manual in Appendix A). In translating a DAME instruction, if a name can not be found in the DAME symbol table, then the DDT symbol table is searched. (These symbol tables are not to be confused with that used by the PDP-11 assembler for PDP-11 symbols. The latter is not saved by the assembler after assembly and is not available to DAME.)

3.2 The Representation of One Main Memory Inside Another

In the next sub-section, 3.2.1, a discussion of the general problem of representing one main memory inside another is presented. Readers interested only in the approach taken in DAME, may skip to the following sub-section, 3.2.2, without loss of continuity.

3.2.1 The General Problem

One of the basic representational issues in simulating one computer inside another is the representation of the main memory of the simulated machine (called the Object Machine or OM) in the simulating machine (called the Host Machine or HM). The importance of this issue arises from the fact that it may have a big impact on the storage requirements as well as the running speed of the simulation. (In this discussion, I shall limit myself to word-oriented machines, i.e. those in which the greatest bulk of memory accesses address words, as opposed to bits, bytes or variable-length blocks.) Two major components of this issue are: (i) The relative word lengths, (ii) The relative sizes of directly addressible memory in the two machine.

Let us denote by W_O and W_H the words lengths, and by M_O and M_H the sizes in words of the object and host machine memories, respectively. (To be more precise, M_H is the size of the portion of the HM memory which may be used to represent the OM memory.) In the usual, and most comfortable, case $W_H \geq W_O$ and $M_H \geq M_O$. This permits an explicit and direct representation of each word of the OM in the HM. If $W_H \geq 2W_O$, then the issue of packing more than one OM word into one HM word comes up. Clearly, if M_O is much smaller than M_H, and main memory cost is not a problem, or, alternately, if the HM has no, or very inefficient, instructions for extracting a field out of an HM word which could represent one OM word, then

the odds are heavily weighted in favor of mapping one OM word to one HM word. It must also be noted that the increased size of storage required to represent the OM memory can also degrade the running speed of the simulation in a time-sharing environment by increasing the page-fault rate or by causing delays in being swapped in by the operating system.

If $M_H < M_O$, one can use a "paged" simulated memory technique, by dividing the OM memory into pages and reading and writing pages as required from a "paging disk" or drum. All the techniques which have been brought to bear to improve the performance of paged systems then become applicable to such a system. If it turns out that the "working-set" of the program under analysis is smaller than M_H, then the performance of this system approaches that of one where $M_O < M_H$.

If $W_H < W_O$, then more than one word of HM are needed to represent one word of OM. In this case, the layout of the OM work must be designed to minimize the overhead of decoding OM instruction operands, and any "tag" bits used by the Hook Mechanism as discussed in Section 3.4 and Chapter 5.

3.2.2 The Representation of the PDP-11 in the PDP-10

In the case of representing the 16-bit 28K PDP-11 in the 36-bit PDP-10 with up to 192K core, initially two -11 words were packed into one -10 word. However, this approach was later abandoned in favor using one -10 word per -11 word and utilizing 18 of the remaining bits in the word to address a list of DAME objects associated with that -11 location. This list, called the Association List (AL) of the location, contains, for example, the hook-objects associated with that location, if any. It is also accessible by the user and may be used to save arbitrary information. However, normally only a small fraction of core locations have a non-empty AL. Therefore, this design decision may be considered wasteful of core. Nonetheless, as will be seen later, in heavily monitored programs, these lists permit much faster access to the monitoring actions associated with a particular location. Thus, in DAME, the low-order 16 bitsof the -10 word are used to represent -11 words, the high-order 18 bits point to the AL, and the remaining two bits are used in the maintenance of input/output sets (Section 3.5).

Only the existing device registers in the peripheral bank are defined; attempts to access undefined locations will result in a "time-out error" on the Unibus and an error trap will occur.

All error conditions are handled just as they are specified in the PDP-11/20 Processor Handbook. The only supported I/O device at present is the TTY. (Recently, 6 relocation registers were added to handle C.mmp programs [WU 72].)

3.3 The Time-Grain of Simulation

This issue has at least as strong an impact on the running speed of the simulation as the representation of the OM memory. The factor which has the major influence on the selection of the time-grain is, clearly, the degree of precision with which one wants to simulate the operation of the hardware. It has already been indicated in Chapter 2 that this should be, at least, at the level of individual instructions. Thus, for example, one would be guaranteed that after each instruction, the state of the memory and the value of the simulation clock would be correct (within the tolerances given in the hardware specifications on which the simulator is based).

The next lower level in the grain of simulation is the "fetch instruction-fetch operands-execute" level, which I shall call "memory cycle level". This level involves, on the average, about 3 to 5 times as many events as the instruction level. If the OM permits intra-instruction interrupts, e.g. after each memory cycle, and if one wants to reflect the timing of these interrupts precisely, then, clearly, one has to design the simulation at this level.

Due to the existence of the so-called "non-processor request" (NPR) interrupts on the PDP-11, although at present no device which can generate NPR interrupts is supported, the simulation has been designed at the memory cycle level. This design decision was also influenced by a desire to permit studies at the processor-Unibus level. The overhead introduced by simulating at this level, as opposed to instruction level, is studied in Chapter 5.

3.4 The Hook Mechanism

The principal mechanism by which the user causes DAME to take some action while his program is running, is the Hook Mechanism. A hook is an object having two user words; the first contains a hook type, and the second a pointer to the list of DAME actions to be taken when the hook is triggered. Hooks may be created, deleted, enabled or disabled dynamically by the HOOK command explained in section 3.6.

There are two categories of hooks: general hooks and addressed hooks. Within each category, there are several types. General hooks are those in which a user-specified DAME action will be taken,

depending on its type, at one of the following points:

1- After every fetch operation (type GF) or,

2- Before every store operation (type GS) or,

3- After every instruction fetch operation (type IF) or,

4- After every instruction completion (type IC) or,

5- After every operand fetch (type OF) or,

6- After every node entry (type NE) or,

7- After every node exit (type NX).

Addressed hooks differ from general hooks only in that they are applicable only when the specified operation (e.g. fetch, store) is performed on an address in a specified range. The types of addressed hooks are:

8- After every fetch from an address in a given range (type AF) or,

9- Before every store into an address in a given range (type AS) or,

10- After every instruction fetched from a given address range (type AIF) or,

11- The completion of every instruction fetched from an address range (type AIC).

To insert a hook, the user issues a HOOK command specifying the hook type, the action to be taken, and if an addressed hook, the address range to which the hook is to be applicable. He can use as many of any type of hook as he desires. Any DAME instruction can be used in these routines.

The types of hooks available in the DAME system, combined with the PROBE command which permits the activation of a DAME routine at a specific time on the simulation clock, satisfy the requirements listed in sub-section 2.5.2, "Triggering of Analysis Actions".

Some Information Made Available by the Simulator

Whenever a hook is activated, the PDP-11 simulator makes available to the user certain information about the state of the

PDP-11 CPU, by storing this information into global PDP-10 symbols. This includes: (i) The address and data associated with the machine cycle which activated the hook, (ii) The operand registers and modes of the current instruction, (iii) Contents of the DATA, ADDR and CONT lines of the Unibus, (iv) The simulation clock, (v) The addresses of the current node object, input set and output set.

The data structures described in Sections 3.1, 3.2 and the above data elements, together with the Execute External (XX) and Evaluate (EVAL) instructions for calling BLISS-10 routines described in Section 3.6.1, satisfy the list of requirements in Section 2.5.1, "Information Requirements of the Analysis System".

3.5 The Node Mechanism

A second major mechanism by which the user causes DAME to collect information about the behaviour of his program, is the so-called "Node Mechanism". The Node Mechanism provides a means by which the user can breakdown all or a part of a program into blocks (called "nodes"), such that each execution of a node (called a "node instance") can be considered as a unit in recording the history of execution of that program. Recalling our requirements about determining data flow among node instances and that any part of the execution must be reconstructible from the recorded execution history, it is clear that we can use the node concept to effect that reconstruction by recreating each instance of each node. To recreate a particular instance X_i of a node X, we need to know all the inputs into X_i. Hence, for this purpose it suffices to record each address from which X_i read something before modifying its contents, and the value read. Let us denote the set of such (address, value) pairs associated with a node instance the "input-set" of that instance. It is easy to see how one can back up arbitrarily far in execution history by restoring the input sets of node instances in reverse chronological order starting with the current node instance. (Note: to simplify references to node instances when the identify of the node itself is not needed, I shall refer to a node instance by its "index" in a particular execution, so that node instance n will refer to the nth node instance since the start of the execution.) We must note here that restoring the input sets of node instances k-p, (k-p)+1,...,k, where k is the current node instance, does not mean that we are restoring the entire machine state which existed when node instance k-p was entered; we are only restoring that part of the machine state which will guarantee an identical replication of the instances k-p through k. Recalling

another functional requirement that we must be able to reconstruct every past machine state, we realize that we must also record the effect of each node instance on the machine state. Such an effect can be represented as a set of (address, old value, new value) triples containing every address where the node instance wrote something (even if the old and new contents are the same) and the contents of that address upon entry and exit from the node instance. Let us call such a set the "output set" of that node instance.

We also note that reconstructing the complete state which existed when node instance k-p was entered, also provides the ability to replicate the execution of the node instances k-p through k, i.e. we do not need the input sets for the purpose of backtracking; the output sets are sufficient. We still do need them however in answering questions about data flow.

One final observation I wish to make is that if an address appears both in the input and the output sets of the same node instance, then its value in the input set and its "old value" in the output set are equal. This means that whenever the two sets contain the same addresses, the first two elements of the triple in the output set (or equivalently, the pair in the input set) are redundant. An empirical study of some input and output sets shows that this redundancy is almost complete, i.e. with very few exceptions, every address which appears in an output set also appears in the corresponding input set. This means that when we restore the input sets of the last n instances in reverse chronological order, we almost always restore the complete machine state which existed just before the n-instance sequence; however we always restore a sufficient part of the machine state to guarantee an identical replication of the execution if a backtrack is requested. (Important note: Here we are neglecting the effects of peripheral devices, such as the setting of status or data registers. These effects constitute communication between two independent processors, i.e. the I/O device and the CPU. DAME does not offer facilities for backtracking over periods in which such communication between two processors occurred. However, such a facility may be programmed by the user and inserted as addressed hooks in such device registers.)

So far, we have not specified whether nodes can be overlapped or nested. In the DAME system, if input/output sets are not being used, nodes may be nested or overlapped, provided they do not overlap at entry and exit points. If input/output sets are being used, overlapped nodes are permitted, provided they do not overlap at entry or exit points. In particular, for example, a subroutine which is called from two different nodes constitutes a part of each node instance in which it is called. If nodes are

Illustration 3.2

Node-object Format

<pred>	<succ>	system words
<class>	<size>	
NODESUBCLASS	0	
<start addr>	<ending addr>	user words
<icount>	<inst. count>	
<ISLP>	<OSLP>	

<start addr> = starting address of node in -11 storage

<ending addr> = ending address of node in -11 storage

<icount> = no. of instructions executed in the last instance of node

<inst. count> = no. of instances of node

<ISLP> = input-set list pointer

<OSLP> = output-set list pointer

nested, input/output sets become much more expensive to build. In the case of un-nested nodes, each address is tagged with a single bit when it is first generated, and in subsequent uses of the same address in the same node instance, that bit prevents it from being entered again in the input or output list; i.e. a search of the input/output set for each generated address is not required to avoid repetition in the input/output sets. With nested nodes however, the single-bit mechanism does not suffice. One needs either a "bit stack" for each address or one has to search the input/output sets of each nested instance for each generated address; both are very expensive to implement. Hence, this particular question may be regarded as an "unsolved problem". In the current implementation, DAME does not permit the use of I/O sets if nodes are nested.

This limitation is analogous to not having a block-structured programming language. While the availability of local variables and dynamic storage allocation with arbitrarily short scopes in a language like ALGOL or BLISS is very desirable in many instances, the same algorithms can be programmed in FORTRAN or APL, which do not have block structure, in very similar ways. Similarly, the unavailability of nested I/O sets did not handicap me significantly in the analysis problems I attacked. I was able to get around the problem by planning my approach in terms of one-level nodes, just as one does in FORTRAN and APL.

3.6 An Outline of DAME Instruction Set

In this section, I shall outline the DAME instruction set and discuss in more detail the instructions particularly useful for monitoring and analyzing of the execution of the PDP-11. Where syntactic descriptions are needed, a BNF-like notation will be used, with "/" denoting disjunction, "<" and ">" delimiting non-terminal symbols and, "[" and "]" delimiting optional operands. The description is intended to be easily understandable and, where there is a conflict between that objective and conciseness and/or terseness, I shall emphasize intelligibility. For those who wish a more detailed description, a document called "Introduction to DAME", which also serves as a User Manual, is included in the Appendix.

DAME instructions can be executed immediately or given a name and saved for deferred execution. The latter are referred to as DAME routines. They can be defined on-line or retrieved from a text file by special-purpose DAME instructions.

The syntax of a DAME instruction is:

<DAME instruction> → <Type-1 instruction> / <Type-2 instruction>

```
<Type-1 instruction> → <operator>(<operand list>)
<Type-2 instruction> → <operator>(<operand list> <action>)
<operand list> → <operand>/<operand list> <operand>
<operand> → <octal integer>/
            <short char. string>/
            <global -10 symbol>/
            <object name>
<action> → <DAME routine name> / <compound instruction>
<short char. string> → <up to 5 characters>
<compound instruction> → (<DAME instruction list>)
<DAME instruction list> → <DAME instruction> /
     <DAME instruction list> <DAME instruction>
```

As can be seen, some DAME instructions take simple operand lists while others (in particular, IF, INCP, WHL, HOOK and ALONG instructions) can optionally take the name of a DAME routine or a compound-instruction (the analogue of a compound statement or compound expression in block-oriented languages) to be executed, as the last operand. All operands of a DAME instruction must be defined prior to the execution of that instruction. Objects, which are not pre-defined by the system, are defined by the Create (CR) instruction (except for DAME routines, hooks and value-trace objects, as described later.) The form @<octal integer> refers to the contents of -11 core location <octal integer> at the time the DAME instruction containing the form is executed.

3.6.1 General Purpose Computation Instructions

DAME provides a complement of instructions corresponding to the usual constructs used in programming, to wit: assignment, arithmetic and logical operations, looping and conditional execution, subroutine calling and I/O. I give an undetailed list of these instructions here in order to convey their basic functions and appearance. A detailed description of their effects is given in Appendix A.

Create object:

```
CR('<obj.name> [<class> <subclass> <size>])
(e.g. CR('A))
```

Delete object:

```
DEL(<obj.id>)
(e.g. DEL(A))
```

Insert in object:

```
IOBJ(<target> <word no.> <value>)
(e.g. IOBJ(A 0 2))
```

Insert indirect in object:

```
IIOBJ(<target> <obj.id> <word no.>)
(e.g. IIOBJ(A B 0))
```

```
Insert in PDP-11 address:
     I(<address> <value>)
     (e.g. I(10000 54))

If-then-else:
     IF(<opd1> '<rel> <opd2> <then-action> [<else-action>])
     (e.g. IF(A 'GT B (IOBJ(A 0 B)) (TOBJ(B 0 A))))

While-do:
     WHL(<opd> <action>)

Incr-from-to-by-do:
     INCR(<var> <from-opd> <to-opd> <step-opd> <action>)
     (e.g. INCR(A 10000 10040 2 (I(A 0))))

Execute DAME routine:
     EX(<routine>)
     (e.g. EX(ROUT1)              execute routine ROUT1)

Push parameter:
     PUSH (<value>)
     (e.g. PUSH(A)                push contents of A)

Pop parameter:
     POP(<obj.id>)
     (e.g. POP(B)                 pop into B)

Return K levels:
     RET(<level count>)
     (e.g. RET(N)                 exit N levels of nesting)

Type out object:
     TOBJ(<obj.id>)
     (e.g. TOBJ(A)                type object A)

Type object indirect:
     TIOBJ(<obj.id>)
     (e.g. TIOBJ(A)               type object pointed by A)

Type PDP-10 symbol:
     TY10(<global variable id>)
     (e.g. TY10(PC)               type contents of program counter)

Type contents of PDP-11 addresses:
     T(<starting address>[<ending address>])
     (e.g. T(10000 A))

Type immediate:
     TI(<literal>)
     (e.g. TI('ABC)               type the char.string "ABC")
```

```
Write disk:
     WDSK(<obj.id>)
     (e.g. WDSK(A)                    write the contents of A
                                      in file USER.DAM)

Write disk indirect:
     WIDSK(<obj.id>)
     (e.g. WIDSK(MNODESC)             write the contents of all
                                      node-objects in file USER.DAM.
                                      Recall that MNODESC contains
                                      a pointer to the node-subclass
                                      master list)

Read disk:
     RDSK(<obj.id>)
     (e.g. RDSK(A)                    read a word into object A
                                      from file USER.DAME.  Read
                                      and write operations on the
                                      same file can not be intermixed
                                      without closing the file.)

Generalized unary operation with assignment:
     UA('<unary op.> <target> <opd>)
         <unary op.> → SUC/PRED/SAL/SIZE/ADDR/NOT
     (e.g. UA('SUC A B)               put in A the address of the
                                      successor of B)

Generalized binary operation with assignment:
     BA('<binary op.> <target> <opd1> <opd2>)
         <binary op.> → +/-/*/<slash>/AND/OR/XOR/!
         where <slash> denotes the division operator "/"
     (e.g. BA('+ A A B)               add B to A)

Execute external routine:
     XX(<PDP-10 routine id> [<param.list>])
     (e.g. XX(TYPLIS 10000)           execute TYPLIS(10000))

Execute external routine and assign returned value:
     EVAL(<target> <PDP-10 routine id> [<param.list>])
     (e.g. EVAL(A TYPLIS 10000) A ← TYPLIS(10000))

Get the value of simulation time and assign:
     TIME(<target> '<scale> '<type>)
          <scale> → MICS/MILS
          <type> → FIX/FLOAT
     (e.g. TIME(A'MICS 'FIX)          Insert in A the simulation
                                      time in microseconds, as an
                                      integer)
```

Plot character:
PLOT(<position> '<char>)
(e.g. PLOT(5 'X) type char. "X" in column 5)

The DAME language was designed to provide a simple syntax in order to minimize syntax errors in the analysis process and to facilitate its translation. It was also intended to be a "low-level" language into which a higher level analysis language, such as the one discussed later in Chapter 6, could be compiled.

While I shall leave the undefined non-terminal symbols and most of the semantics of the above instructions unelaborated, a few explanations are in order. Wherever a numeric argument is expected, if a name is supplied, its contents are taken. The non-terminal <target> denotes the name or address of an object into which the assignment is to be make. The syntax of the generalized unary and binary operations with assignment is admittedly very awkward, but it permitted me to save some code in interpreting the operands for each operation.

In addition to these instructions, since the fundamental data structures used by DAME are lists, there is a set of list manipulation facilities. Some of these are provided by POOMAS and are accessible via the AX and EVAL instructions listed above. These are routines for creating, deleting and maintaining lists. DAME provides facilities for taking the union, intersection and set difference of two lists and assigning the result to a third list, in a syntax similar to the preceding instructions. It also offers a unique "Search List" instruction whose syntax is:

SLIST(<target> <list id> <search spec.>)
<search spec> → <action>

which works as follows: DAME pushes the address of the first element of the list <list id> on the "data stack". (The Push and Pop instructions listed above operate on this stack. There is a second stack, called the "monitor stack", used for DAME routine calls.) Control is then passed to the DAME instructions specified in <search spec.>. In the <search spec.>, the user must obtain the stacked address with a POP instruction. He can then perform arbitrary computations, preferably without further manipulating the stack. If he wishes to end the search, he PUSHes a 1 by DAME, in which case the stack will be popped by DAME, the address of the current element in the list will be stored in <target> and the instruction execution terminated. If the user wishes to continue the search, he can PUSH a 0. In this case, after the stack is popped, if the end of the list is reached, DAME will insert a 36-bit -1 in <target> and terminate the instruction. Otherwise, the address of the next element of

the list will be PUSHed and the cycle will be repeated. For example, the instruction

```
SLIST(A LISTA (POP(B)
               IIOBJ(C B 3)
               IF(C 'GE 5
                    (PUSH(1))(PUSH (0)))))
```

would search LISTA for an element whose fourth user word contains 5. If such an element is found, its address is returned in object A; otherwise A will contain a -1.

3.6.2 Execution Monitoring and Analysis Instructions

In this section, the subset of the DAME instruction repertoire which perform the functions essential to providing the wide range of facilities desirable in a general purpose execution analysis facility is described. The style of the exposition is again narrative and informal to give a good intuitive understanding of the primitive operations and data structures involved. Starting with the instructions for inserting hooks, defining nodes and creating input/output sets, I shall describe instructions for searching input/output sets, restoring node instances, "instant replays", monitoring specific paths of control flow, automatically collecting the last k values of a location and addressing them via an operation similar to indexing, as well as the instructions for typing out node objects and node instances which have been mentioned before.

The Hook Mechanism, described earlier, is used to insert hooks to perform the user-specified actions at user-specified times. Instructions for manipulating hooks are:

```
HOOK(<hook-type><action> [<address range>] <hook name>)
(e.g. HOOK('IC (TOBJ(A)) 'HIC)   Type the contents of A
                                 after every instruction)

DEL(<hook name>)
DISAB(<hook name>)
ENAB(<hook name>)
(e.g. DEL(HIC), DISAB(HIC), ENAB(HIC))
```

(Note: Brackets [,] indicate optional operands.)

These will insert, delete, disable or enable, respectively, a hook named <hook name>. The <address range> is only required for addressed hooks.

Creation of Nodes and Input/Output Sets

The Node Mechanism, also described earlier, can be evoked in one of two ways: via the NODE instruction or via the NTR instruction. The syntax for the former is:

NODE(<address range><node name>)
(e.g. NODE(20000 20100 'NODEA))

The execution of this instruction will cause a node-object of name <node name> to be created. The format of the user-words of a node-object is given in the following figure. Each node-object contains, among other data, a pointer to each of two lists its input-set list (ISL) and its output-set list (OSL). If the node has not been executed as yet, these lists are empty. An input (or output) set consists of a list of ten-word objects. An (address, value) pair is inserted into each word, starting with the first word of the first object. When and if the first object is full, a second object is created, and so on. The list head contains one user word which contains the index of the first empty word in the last object. All unused words contain zeros. The high order bit of each word contains a 1 if only a byte was accessed, 0 otherwise. (The only variation to this rule is in the case of the Processor Status word PS. Since the PS is essentially bit-addressable, an indication of which bits were read or modified is needed. To do this, we can take advantage of the fact that only the lower 8 bits of the PS are usable by the user. When the PS appears in an input/output set, a bit mask in the upper byte of the "contents" part indicates which bits were accessed. However, this feature is not implemented and the PS is treated like any other address.) The PS and all the general and device registers are represented by their console addresses. The format of ISL's and OSL's is given in Figure 3.3.

To provide for more flexibility in the use of I/O sets, separate instructions to initialize and build I/O sets have been provided. Since the building of these sets adds quite a bit of overhead to the execution, I have found it useful to prepare a standard DAME routine in a text file which I can evoke only when I wish to construct I/O sets. The instructions provided for this purpose are IIS(), BIS() and OIS() to initialize, build and close input sets, IOS(), BOS() and COS() to perform the same functions for output sets. (The parenthesis pairs indicate empty parameter lists and are required by the syntax of the language.)

For example, the following hook causes the initialization of a new input-set at each node entry.

HOOK('NE (IIS()) 'HNE)

Illustration 3.3

INPUT-SET LIST (ISL)
for a Node

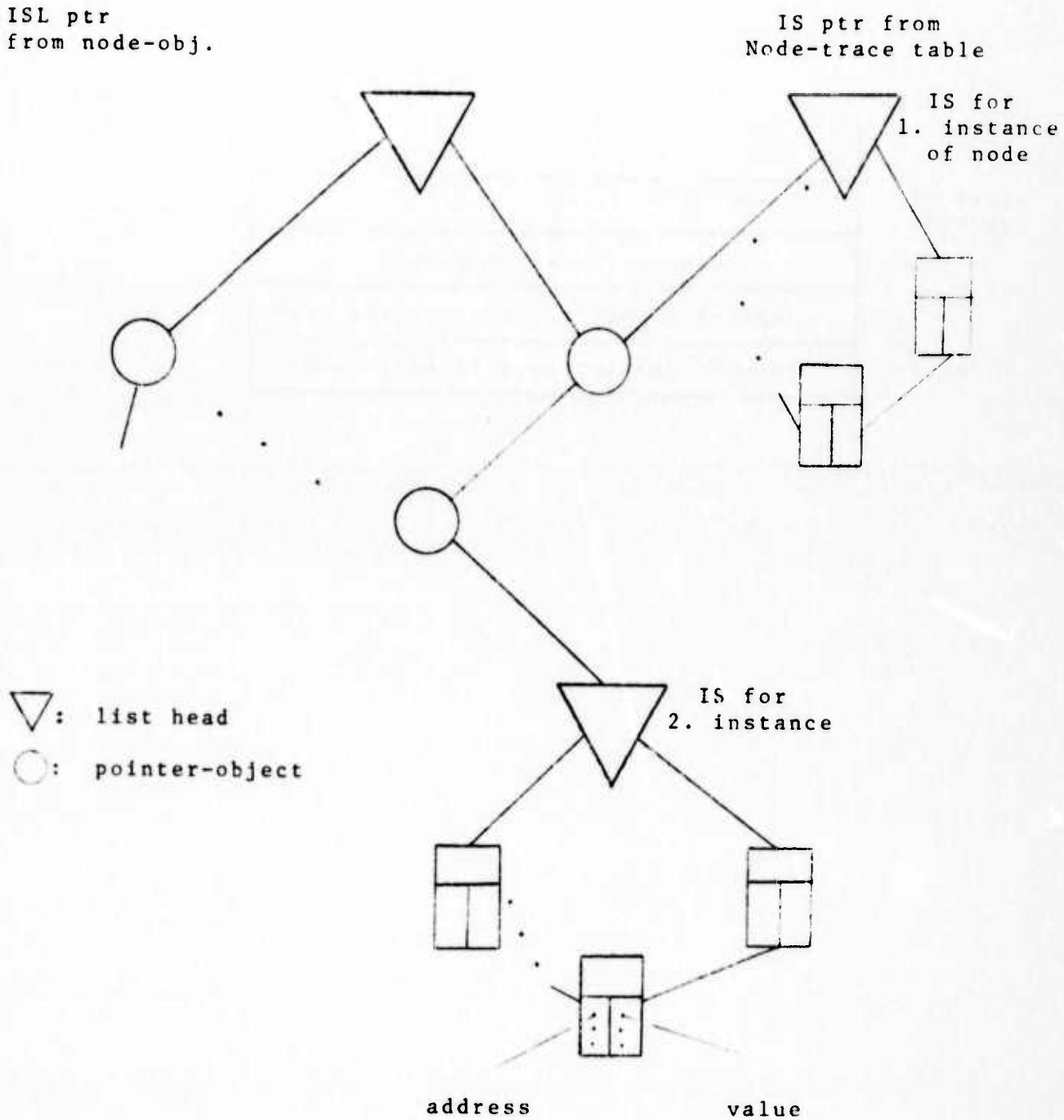
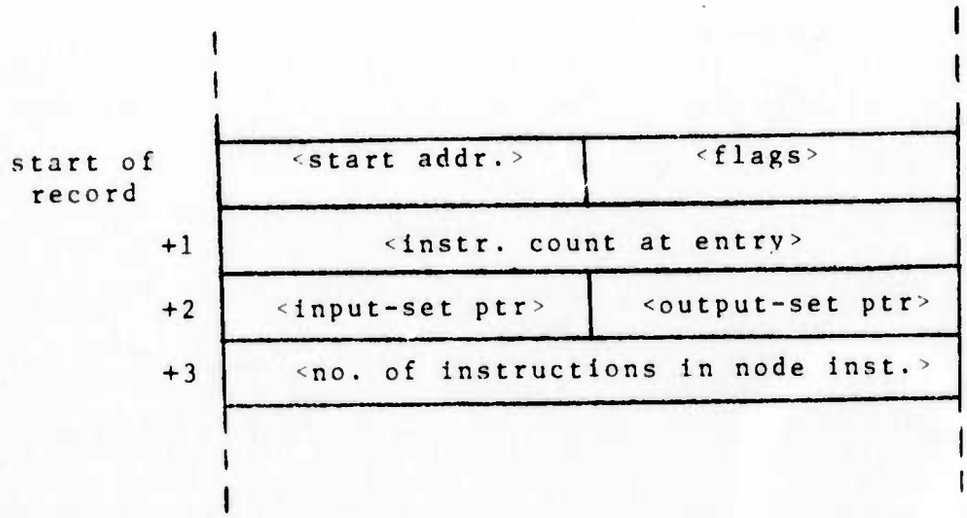


Illustration 3.4

NODETRACE Table Record

for a Node Instance



Here HNE is the name of the hook object and can be used to disable or enable the hook later, e.g. by the instruction

DISAB(HNE) or,
ENAB(HNE).

The definition of nodes can be defaulted by using the Node Trace, NTR(), instruction. In this case, nodes are defined by DAME by monitoring the control flow. When it varies from sequential flow for any reason or when an unsuccessful conditional branch is executed, the current node instance is terminated and a new node instance is entered. If the new instance is the first one for that address, a new node object is created. Thus, in the default mode, nodes most often terminate with a branch or subroutine call instruction. I/O sets are constructed in the same manner as when the NODE instruction is used.

Searching of Executed Nodes and Input/Output Sets

Given the formats of the node-objects, the I/O sets and the node-trace table described earlier, the user can extract whatever execution information he needs by using the language facilities, in particular the list-processing operations. However, to facilitate most common types of analyses, a set of instructions intended specifically for searching these data structures is provided. These instructions are: Find Input Set (FISET), Find Output Set(FOSET), Find Node Object (FNO), Find Node Instance (FNI), Find Value (FVAL), Find Value Indirect (FIVAL), Playback Values (PLAYB), Restore Node Instance (REST), Replay Node Instance (RPLAY), Type Node Instances (TNI) and Type Node Objects (TNO). I shall now describe each of these instructions in detail.

The Find Input Set (FISET) instruction attempts to find an input set satisfying some user-specified conditions. The specifiable conditions are the identification of the node whose input sets are to be searched, the starting point and the time direction of the search (i.e. forward or backward in execution history), and a predicate which should be applied to each input set. The language for specifying the predicate was of some concern, since the predicate could be arbitrarily complex and it was undesirable to design a whole new language for this purpose. The technique finally adopted was that described under the Search List (SLST) instruction, i.e. to let the system help the user find each input and access the value contained in it, and to let the user perform arbitrary computations on them, and then tell the system whether he wants to continue or stop the search, using the facilities already implemented in the system. I shall now desreibe this procedure. The syntax of the Find Input Set instruction is:

FISET(<object id> <node spec.> <search spec.>
[<direction> [<starting index>]])

Let us ignore all the operands except <search spec.> for the time being. <search spec.> must be a DAME routine name or an explicit instruction sequence (similar to a "compound statement" or "compound expression" in some programming languages). Before the <search spec.> is entered, the system locates and internally PUSHes the address of the next input set to be searched (PUSH and POP were described in the preceding section). The user must obtain this address by a POP(A) instruction, where A is some object name, which puts the address of the input set to be searched in object A. Then the contents of address K in that input set can be extracted and saved in some object B, by the Find Value instruction, as FIVAL(B K A). The user can, in this manner, obtain the contents of any address in the input set pointed by A, and perform calculations on them using the language facilities. If he is finished with the search (e.g., he has found the input set he is looking for), he PUSHes a 1; otherwise he PUSHes a 0. After the last instruction in <search spec.> has been executed, the system will POP the stack. If the value is 0, then if the end of the node trace has been reached, it will insert a -1 in <object id> and will terminate the FISET instruction. If the value is 0 and the end of the node trace has not been reached, it will push the address of the next input set to be searched, proceeding in the direction specified by <direction> and re-apply <search spec.>. If the popped value is a 1, the index of the node instance just searched will be inserted in <object id> and the instruction will be terminated. Thus, after the FISET instruction, <object id> will contain either -1, which indicates that no input set satisfying the specifications was found, or it will contain the address of the first acceptable set.

To illustrate the use of this instruction, suppose at some point in the execution we wish to find the most recent input set where the contents of location 1000 equal the contents of location 2000, and put the address of that input set into some object D. To do this we shall need three more objects (in fact, we could get by with one by using the same object for various purposes but we shall not do so here). The following instructions create these objects and perform the required search:

```
CR('A) CR('B) CR('C) CR('D)
FISET(D '* (POP(A)
          FIVAL(B 1000 A)
          FIVAL(C 2000 A)
          IF (B 'EQ C (PUSH(1)) (PUSH(0)))
            ))
```

The symbol '* for <node spec.> indicates that all nodes are to be searched. The syntax of the IF instruction is:

IF(<obj.id> <relation> <obj. id> <then-case>[<else-case>])

The Find Output Set (FOSET) instruction works exactly in the same way as FISET, except that output sets are searched.

The Find Node Object instruction, whose syntax is FNO(<obj. id> <-11 address>), inserts in <obj. id> the address of the node object associated with <-11 address> if such an object exists. Otherwise a -1 is inserted.

Find Node Instance, FNI(<obj. id> <node id> <n> [<starting index> [<direction>]]), will similarly insert in <obj. id> the index of the nth instance of <node id> searching the node trace in the <direction> specified starting from <starting index>. The default values for the two optional operands are: "the current node instance" and "backward", respectively.

Find Value and Find Value Indirect are used to extract the value associated with an address in an input or output set where the I/O set address is given in the instruction, and where the I/O set is pointed by the object given in the instruction, respectively.

The Restore to Node Instance, REST(N), instruction moves backward in execution time, restoring the input sets of node instances until index N is reached; e.g., if the current node instance is the Kth node instance executed, REST(N) would restore the last (K-N+1) input sets.

The Replay Node Instances instruction RPLAY(<starting index> [<ending index>]), will cause the restoration of the input sets of the node instances between the specified indices. The simulation time is also restored. The instances whose input sets have been restored are then re-executed. Upon termination of the last instance the environment in which the RPLAY instruction was issued is re-established.

The Type Node Instances instruction TNI([<starting index>] <count>) types the node trace entries for abs(<count>) instances, starting at <starting index>, and moving forward in time, if <count> is positive or backward if <count> is negative, where abs(x) denotes the absolute value of x.

The Type Node Objects instruction, TNO(<address1> <address2> ...) types out the node objects associated with the specified addresses.

Detecting Specific Paths of Execution

I would now like to describe the instruction ALONG, whose syntax is:

ALONG(<path> <action>)
<path> → <node id> / <path> <node id>
<action> → <DAME routine name> / (<instruction sequence>)

Suppose we have defined nodes N1, N2, ..., N7. Then, the instruction

ALONG(N1 N5 N7 X)

would cause the action X to be taken if the current node is N1, or if the last two nodes have been N1 and N5, or if the last three nodes have been N1, N5 and N7, in that order. In short, the specified <action> is taken whenever the flow of control could be following the specified path. The ALONG instruction is, as are all DAME instructions, executable through every type of hook. Hence it provides a convenient facility for taking selective action (e.g., tracing) as a function of the locus of control flow.

Collecting and Accessing Precious Values of a Location

Finally, a mechanism for automatic collection of the previous values of a location and for accessing those values is worth mentioning. The first action is accomplished through the use of two instructions. The first is the Initialize Value Trace, IVT(<-11 address> <n> <obj. name>), which creates an object named <obj. name> of a special subclass and large enough to hold <n> previous values of location <-11 address>. The second instruction is the Value Trace Hook, VTH(<-11 address>), instruction which causes the monitoring of values stored into the location <-11 address> and maintains the last <n> such values in a circular buffer in object <obj. name> created by the IVT instruction. Then, at any point in the execution, the Kth previous value of <-11 address> is obtainable by a binary operator #, as

BA('# B <-11 address> K).

The instruction BA(<opr> <target> <opd1> <opd2>) is the generalized "Binary Operation with Assignment" instruction and performs the operation: <target> ← <opd1> <opr> <opd2> in infix notation. Thus the above instruction would insert in <-11 address the Kth previous value of <-11 address>. If K is larger than the number of values declared to be kept in the IVT instruction, an error message will be typed and no assignment will be made. If K values have not yet been assigned to <-11 address>, then a special code larger than 2↑16 will be stored in B.

3.7 Various Design Issues and Unimplemented Ideas

In this section, I shall discuss some design issues which arose in the course of the development of DAME. Most of them are related to improving the execution speed of the simulation and decreasing the monitoring overhead. I shall also outline some ideas which have not been implemented mainly because they would not contribute significantly to the research aspects of this project.

3.7.1 Representation of -11 Core and the Design of the Hook Mechanism

Since the representation of the PDP-11 core and the Hook Mechanism lie at the heart of the implementation plan, these two points are worth re-pondering and alternative implementations worth considering.

As was mentioned, an earlier implementation of the simulator packed two -11 words into a -10 word, one into the low-order 16 bits of each of the lower and upper halves of each 36-bit -10 word. In that implementation, the high-order two bits of each -10 halfword were used to indicate the presence or absence of monitoring actions associated with the fetch or store of each data word (e.g. a word fetched or stored by an instruction) or, with the fetch or completion of an instruction. The monitor actions themselves were located via a table look-up on the particular address involved. A seperate table was used for each of data fetch, store, instruction fetch and instruction completion operations. This design makes possible a substantial saving in the core requirement, approximately ((28K/2)-n), where n is the number of locations for which a hook exists. The essential price paid for this storage saving is the overhead of the table look-up procedure. Assuming that approximately 1% of the locations are hooked and a binary search is used, about 8 comparisons are needed to locate the monitor action pointer associated with a particular address. Further assuming that one address involved in every instruction has some monitor action associated with it, this overhead is roughly equivalent to twice the overhead of decoding the op-code of an -11 instruction. In addition to the monitoring actions associated with particular addresses, there are those due to the so-called "general hooks", i.e. actions to be taken at every fetch, or every store etc. Thus, there already is substantial overhead due to monitoring. So, the decision to map one -11 word into each -10 word and use the left half of the -10 word for a pointer to associated monitor actions was intended to avoid further degradation in the monitoring overhead, but exactly how much is gained in response time in a time-sharing environment is not clear since the larger core requirement delays

the swapping-in by the operating system scheduler. When the word lengths of the object machine and the host machine are equal and one would like to use a one-to-one mapping, a scheme proposed by Bernard Lang, [LA 72], called "Lambda-monitoring", can be used. In this scheme, since there are no additional bits available to indicate the presence of associated monitor actions, one inserts a special bit pattern (called "Lambda") into the word when one wants to associate monitor actions with it. Then, at every fetch or store, one compares the contents of the address being accessed with the bit pattern Lambda. If they are found equal, this is taken as a signal that there may be some monitor action associated with that address. Then, tables set up for this purpose are searched, just as in the earlier schemes. If an entry for that address is found, the monitor action indicated by the entry is performed. The table entry also includes the actual contents of that location. If no entry for that address is found, no action is taken and the execution is permitted to continue.

This scheme is clearly very similar to the scheme used by current debugging systems which insert a trap instruction into any instruction address where the user wants to put a breakpoint. The "Lambda-monitoring" scheme simply extends this technique to apply to data elements as well as instructions.

In using such a scheme, clearly, "bugged" locations must be write-protected from the user; i.e. the data to be stored into such a location must in fact be trapped and re-routed to a special register holding the actual contents of that location. That register is the same one whose contents are fetched upon a fetch operation on the bugged location. The first requirement implies that prior to every store operation, the current contents of the store address must be fetched and compared with Lambda.

I shall have more to say about this technique in Chapter 8, when I go into the implementation of monitoring features in microprogram or hardware.

3.7.2 Scheduling with Look-ahead

One of the main bottlenecks in the simulator is the event scheduling process. As was mentioned, the time-grain of the simulation is at the memory/register access level. The particular simulation package which is used is a general-purpose simulation package, in which an Event Notice is created for each event to be scheduled showing the time of activation and the process to be activated. After each event, the scheduler consults the event calendar and activates the process indicated by the first event notice having the earliest time of activation. In our case,

since there are no simulated devices other than the TTY, there are usually only two processes which receive and surrender control: the CPU and the Unibus. Further, the two are never active simultaneously in simulated time. While this design is a clean and consistent one, permitting the addition of new devices to the Unibus in an easy way logically quite similar to adding them to the real Unibus, and also permitting studies on bus utilization, timing of signals between devices on the bus etc. to be done very naturally, it is also quite expensive in terms of scheduling overhead due to the event notice preperation, placement and searching of the simulation calendar.

A technique which can be employed to reduce this overhead is what may be termed a "look-ahead" technique, in which the CPU checks the simulation calendar before it releases control to the scheduler. If it finds no events scheduled (e.g. an I/O device activity), rather than releasing control to the scheduler which would activate the Unibus next, the CPU performs the core access function itself, perhaps by calling a "routine" version of the Unibus (as opposed to a coroutine version) which performs an identical function as the coroutine version without the co-routine jump statements.

Some measurements on the gain in simulation speed through this technique is reported in Chapter 5.

3.7.3 "Blow-up" Representation of the Processor Status Word

Another technique by which the speed of the simulation may be increased is reducing the amount of individual bit manipulation in the handling of each PDP-11 instruction since this is a very slow operation in the PDP-10 (at least, in our model). A good candidate for this case is the modification of the Processor Status word (PS), since most instructions modify one or more bits in this word. Further, each bit must be computed and set seperately. Since the PS is affected by most instructions, this causes a good bit of overhead.

This problem can be alleviated to a certain extent by representing each of the six fields of the PS by a seperate word. However, caution must be taken that, in case the user program explicitly addresses the PS, then the result of the read or write operation is reflected properly on the Unibus lines and the words representing individual PS fields.

3.7.4 "Compilation" of Decoded -11 Instructions

As it is designed, the simulator re-interpretes every PDP-11 instruction every time it is executed. In particular, the extraction of the op-code, the operand addressing modes, the operand registers and the selection of the particular simulator routine to be called, causes considerable loss of efficiency in each re-interpretation of an instruction. What this suggests is a "compilation" of each executed PDP-11 instruction into PDP-10 code tailored specifically for that particular -11 instruction, in which all variability has been eliminated. This would provide for much more efficient execution of that -11 instruction subsequently. (The concept of processors which can execute both compiled and interpretive code has been implemented in various systems, e.g. PDP-10 LISP. Also see J. Mitchell for a good discussion of this topic [MI 70].)

There are several problems which must be resolved however. One is the fact that there will be considerable overhead associated with the compilation itself; therefore, instructions which will be executed fewer than some number, n, times should not be compiled, where n is a function of the actual overhead of compilation versus field interpretation. However, in general, we do not know beforehand the number of times each instruction will be executed. Hence, it is difficult to tell which instruction to compile and which not to compile. One heuristic rule which can be used is that if an instruction is used a second time, it is probably a part of a loop or a common subroutine etc. and hence its chances of being used again are good. Therefore, a reasonable approach may be to compile an instruction the second time it is used. There will clearly be some waste due to the compilation of instructions which are executed exactly twice or even those which are executed, say, three or four times. This parameter, namely, the number n, can be examined more thoroughly after the compilation process has been implemented; it may well turn out that this number varies as a function of the instruction class, e.g. a simple unconditional branch may not be worth compiling at all, whereas a double-operand instruction may be worth compiling after its first execution.

It is clear however that the simulator has to be able to execute both forms of PDP-11 instructions, i.e. the "uncompiled" PDP-11 machine instruction and the "compiled" version, which in the ultimate, is a sequence of PDP-10 machine instructions associated with the particular -11 instruction location.

Another question which must be resolved in order to use this technique is how to associate the -10 code with the appropriate -11 instruction address. One solution may be to insert the -10

instructions associated with a particular -11 location into an object and to use a table of size k, containing pointers to these objects, where k is the size of -11 memory containing instructions. The overhead required to locate the required -10 code must be minimized to make this technique worthwhile.

In the design of DAME, there is a particular feature, namely the Association List for each core location, which solves this problem very naturally. One can insert the object containing the -10 instructions for a particular -11 location as the first element in the Association List of that location. If more generality is desired, one can introduce a new subclass, called "PDP-10 code subclass" and insert an object of that subclass anywhere in the Association List. However, this will of course increase the search time. The use of association lists for this purpose also obviates the need for the large table required by the first technique.

Finally, in this connection, we must note a problem with self-modifying programs, namely that if a particular instruction is modified during the course of execution, its old "compiled" version must be deleted and a new decision has to be made as to whether the new version should be compiled. In fact, if a particular instruction will be changed frequently, it probably should not be compiled.

3.7.5 Further Compilation of DAME Instructions

Another area worth considering for improved efficiency is that of further compilation of DAME instructions into -10 code. This is particularly true for heavily monitored programs. At present, DAME instructions are only "assembled", i.e. the DAME op-code and symbolic operands are replaced with the -10 addresses of the routine to execute that instruction and the addresses of the operands, respectively. Such things as determining the number of operands and certain kinds of type and size checking are done at run-time. It is possible to do a large part of this at routine-definition time since object size, class and subclasses are declared when the object is created. This would not be possible, however, for indirectly accessed objects.

3.7.6 A "Limited-Run Complete-Trace" Feature

As was described earlier, backtracking to a particular instruction n is implemented by restoring in reverse chronological order, the input sets of node instances until the one including the instruction n is restored, and then executing the instructions preceding n in that node instance. Backtracking has been implemented in a different way by, at least, one more worker, Ralph Grishman,

in the AIDS system at NYU Courant Institute of Mathematical Sciences. The following description of the implementation of this mechanism is taken from R. Stockton Gaines' thesis:

"... The back-up mechanism mentioned above is original with Grishman, and is sufficiently interesting to warrant a detailed description of how it is accomplished. AIDS keeps four tables for this purpose; let us call them R1, R2, S1 and S2. As AIDS is interpreting the user's program it goes through the following process. At some point it saves the state of the machine registers in R1, and after that each time the user's program stores a new quantity into a location in memory, the previous quantity at the location is saved in S1 together with the address which is being changed. When S1 is full, the registers are stored in R2, and execution continues with AIDS saving the previous values and addresses to which stores are made in S2. When S2 is full, the process starts over again with R1 and S1, and so on. When the user issues a request to back up, AIDS fetches the most recent item from S1 or S2 and puts it back where it was originally. It then puts back the next most recent, and so on, until it has put back the first quantity saved after the next to last time the registers were saved. At this point it can restore the registers to the values they had the next to last time they were saved, and AIDS can reexecute the program from that point to the interrupt at which the back-up was requested, since the program and its storage are now in the same condition they were when the program reached that point for the first time..."

While I believe that the node mechanism and the input/output set concept of DAME have significant advantages over this method in terms of storage requirement and the ease with which the collected information may be used in data flow analysis, there are times at which the user would like to see a complete trace of certain portions of his program. At present, this can be done in DAME by attaching general hooks to fetch, store and instruction completion events to type out the required information. Alternatively, if the number of instructions to be thus traced is small, each instruction can be declared a node, in which case the node mechanism will construct the input and output sets for each instruction. Nevertheless, it may be desirable to have a "detailed trace" mode in which every memory and register access is recorded in a "trace object". This would be useful, for example, in directly answering questions like "What was the second value assigned to X in node N?", or "What was the value of X at instruction I?", without the restoration of the required input sets etc. However, such a facility would have to be used in a highly selective and judicious manner since it would require a great deal of storage and CPU time overhead.

CHAPTER 4

ILLUSTRATIVE EXAMPLES OF SOME APPLICATIONS OF DAME

In this chapter, I shall illustrate the main features of DAME through a set of examples of its application. A modest familiarity with the architechture of the PDP-11 will be helpful, and assumed, in the examples. Assembly language notation will be explained as necessary.

Five examples are given. The first one demonstrates the nodes and the input and output sets of a program, as well as the mechanics of loading a PDP-11 program, inserting hooks, initialization of -11 core and the initiation of execution. The second example demonstrates the construction of a node transition matrix M, whose element M(i,j) is the number of transitions from node i to node j since the beginning of the execution. In the third example, the lower bound for the elapsed time (in terms of executed instructions) required to execute a recursive program given an unbounded number of identical processors, is calculated; in fact, the structure of the execution tree is determined and displayed. The fourth example demonstrates the construction of a set $X_i(N,M)$, each element of which is a triple (a_1, a_2, a_3) where a_1 is an address which is in the intersection of the input set of the ith instance of node N and the output set of the preceding instance of node M. If node M has not been executed between the (i-1)th and ith instances of N, then X_i is empty. a_2 is the value read from location a_1 by N and a_3 is the value of a_1 at the exit from the preceding instance of M.

These four examples are intended to provide illustrations of dynamic analyses of control flow, data flow and performance, which are the main types of analyses for which a system like DAME is suitable. They will all use the same PDP-11 program to demonstrate the various analysis techniques. For the purpose of simplicity in exposition, the chosen program is a small one -- a one-page "quicksort" routine. Its code is given and explained in Example 1.

The fifth example is not one for which DAME is particularly suitable, but is included here to show that even in cases which would "strain" a simulator-based software monitor system, one is able to make useful analyses by exercising some intelligence in its use. This example deals with collecting instruction mix

and addressing mode usage statistics on several PDP-11 programs. The collected statistics, while they are interesting and possibly useful in their own right, were used to project the running times of the same programs on a PDP-11/40 and -11/45.

Example 1. Nodes and Input/Output Sets of a Quicksort Program

As a first example, let us consider the PDP-11 assembly language program QUICKSORT, whose text is given in the next illustration. (For a specification of the DEC assembly language see [DEC 71].) The program implements a simplified version of the "quicksort" algorithm as given by Knuth in [KN 73]. The code given here was compiled by the BLISS-11 compiler [DEC 73] to be assembled by the MACX11 assembler. To explain briefly some of the notation in the assembly language: # denotes immediate operands, Ri means register i, SP is the stack pointer (register 7), @ denotes indirect addressing, -(K) denotes automatic decrementation of register K before its contents are used, and (K)+ denotes automatic incrementation of register K after its contents have been used. All integers are in octal. The syntax of double-operand instructions is:

<opcode> <source-operand>, <destination operand>

(All integers are in octal.)

The program consists of two parts: a recursive subroutine called QSORT located between (relative addresses) 0 and 166, and the main program between 170 and 204. The main program expects two integers in registers 0 and 1, which are to be the bounds of the core locations whose contents are to be sorted. It simply pushes these parameters on the stack (which grows downward from its initial value of 1400) and calls the subroutine QSORT. This subroutine works as follows:

It uses R1 and R2 to point to the lower and upper bounds, respectively, of the vector to be sorted. If R1 is greater than or equal to R2, there is no sorting to be done; hence it returns. Otherwise, it compares the elements pointed by R1 and R2. If no exchange is necessary, R2 is decremented by 1 and the process is repeated. After the first exchange R1 is incremented by 1 (Note: Since sorting is done in units of words, the addresses are really incremented by 2). Comparison with the element pointed by R2 and incrementation continues until another exchange occurs, at which point R2 is decreased again. The sorting goes on this way, "burning the candle at both ends", until R1 and R2 point to the same element. During this process, the value which was initially pointed by R1 has been exchanged everytime the direction was switched. When R1=R2, this value will have found its final position:

i.e., the position it must have in the completely sorted vector. (The interested reader can convince himself of this.) Further, this element now divides the vector into two parts, namely, that to its left and that to its right. These two parts, which Knuth calls "subfiles", can be sorted with the same procedure. Hence, QSORT then calls itself twice, to sort first the left subfile and then the right subfile.

The -11 code given in Illustration 4.1 was produced by the BLISS-11 compiler and the comments, preceded by the symbol "!", were inserted later by hand.

In this example, I shall load the -11 program, which is stored in a file named QSORT, initialize a vector of 40 elements to be sorted, set the default mode for node definition and, at every node-exit, type out the node-object and the current input/output set for the first five node instances. Then I will let the program run to completion. The monitor instructions which will be used to do this are contained in a file called DEMO1, which is listed in Illustration 4.2. In this file, there are two routines: DEMO1 and TIO (for Type I/O Sets). DEMO1 loads two copies of the Quicksort program, one starting at location 20000 (this copy is the one which is executed) and another starting at location 30000 (this copy will be used as data: a small part of it will be sorted.); it initializes registers R0 and R1; loads another routine DEFIO from a file called DEFIO and executes it; defaults the node definition; creates an object C to be used for counting to 5; initializes it to 0; inserts a hook to call the routine TIO at every node exit; gives control to the PDP-11 starting at location 20170, the address of the main program. (All "relative addresses" in this example are relative to 20000.)

The routine DEFIO (listed at the bottom of illustration 4.2) is a standard routine for constructing input/output sets. It works as follows:

The symbols CURNOBJ, CISP and COSP used in TIO are global PDP-10 variables which point to the current node-object, the current input-set and the current output-set, respectively. (As a practical matter in the use of DAME, if the monitor routines to be used turn out to be long or if we aren't sure they are correct, it is a good idea to prepare them as text files and load them at run-time rather than define them on-line, during the analysis session.) The format of type-out for objects and lists is: The words of an object are typed between slashes. If a word is not zero, it is typed as <left half>,,<right half>, otherwise it is typed as 0. Thus, node-objects are typed out as:

<starting loc.>,,<ending loc.> /<no. of instr. in it>,,
<no. of instances> /<input-set ptr>,,<output-set ptr>

Lists are typed as [<object> <object>...]. Certain objects, called "rep-objects", which are an artifact used for implementing hierarchical list structures and member ship in multiple lists, contain only pointers and are typed as → followed by the pointed object. I/O sets are simply lists of ten-word objects each containing up to ten <address>,,<value> pairs. Unused words contain zeros. Thus, an I/O set containing up to 10 (address, value) pairs is typed out as:

[<address>,,<value> /<address>,,<value> /.../0 /0].

An I/O set containing more than one 10-word chunk is typed out as:

[<first 10 words> → <next 10 words> → ...].

(Registers 0 through 7 are represented by their console addresses 177700 through 177707 respectively, and the processor status word by 177776.)

Illustration 4.3 shows the protocol for this example. User-typed portions are underlined. The comments in small type were entered later and are not a part of the protocol.

It inserts the hooks named HIOS (Initialize Output Set) and HIIS (Initialize Input Set) to be activated at node entry (i.e. hooks of type NE). These operations could have been done with one hook but this method permits either one to be disabled and enabled without effecting the other. These routines issue the IOS() and IIS() instructions respectively. To build the I/O sets during the node instance, the hooks named HBOS (Build Output Set) and HBIS (Build Input Set) are inserted to be activated at every operand store and every operand fetch, respectively. These routines issue the BOS() and BIS() instructions to maintain their respective sets. At node exit, the input and output sets must be closed. This is done by the hooks named HCIS and HCOS, by issuing the CIS() and COS() instructions respectively. One final problem is that in case the entire -11 program is not covered by nodes, the hooks HBIS and HBOS must be turned off at exit from a node and re-enabled at entry into a new one. Thus, DEFIO initially disables these hooks, and inserts the hooks named HENB and HDISB which are activated at node entry and exit respectively. They call the routines ENAB and DISAB to perform their functions.

While this procedure for building I/O sets is rather long and elaborate, it is more efficient and flexible than automatically

maintaining the I/O sets. Further, by preparing it as a file and simply calling it when required, it can be used easily.

The routine TIO tests C. If it is less than 5, the following is done: A carriage return-line feed (CRLF), and the character string 'NODE:' is typed (in DAMĒ, a character string can not normally exceed 5 characters and it preceded by a single quote); the current node object (pointed by the global CURNOBJ) is typed by the Type Indirect instruction TIOBJ which also types a CRLF; this is followed by the text 'INPUT-SET '(typed in two pieces), the current input set (pointed by CISP), the text 'OUTPUT-SET ' and the current output set (pointed by COSP). The counter C is then incremented. If initially C is not less than 5, the hook HTIO which activates TIO is disabled.

The protocol shown in Illustration 3, also illustrates the instruction PLAYB. In this case, PLAYB (177701 3), recalling that in the PDP-11 177701 is the console address of register 1, types out certain information about the state of the program at the 3 most recent node entry or exits where register 1 appeared either as input or output.

The message '---HALT AT 20206' following the last I/O set is typed out by the simulator when the halt instruction is encountered.

Illustration 4.1

```
Relative            In-line
 Addr.     Code     Data                                                             Comments

                              subroutine QSORT:
000000    010146                             MOV    R$1,-(SP)    }
000002    010246                             MOV    R$2,-(SP)    }save regs. on stack;
000004    010346                             MOV    R$3,-(SP)    }
000006    012700   000001                    MOV    #1,R$0       !set flag=1 initially;
000012    016601   000012                    MOV    12(SP),R$1   !get 1.parm into R1;
000016    016602   000010                    MOV    10(SP),R$2   !get 2.parm into R2;
000022    020102                             CMP    R$1,R$2      !lower bound ≤ upper bound?
000024    002402                             BLT    L$4          !yes, go do sort;
000026    005000                             CLR    R$0          !no, set value reg.=0;
000030    000453                             BR     L$2          !go to exit prologue;
000032    020201                    L$4:     CMP    R$2,R$1      } if R2 ≤ R1 go to U$1;
000034    003430                             BLE    U$1          }
000036    021112                             CMP    @R$1,@R$2    } if left elm. ≥ right elm.
000040    003415                             BLE    L$10         }    go to L$10;
000042    011103                             MOV    @R$1,R$3     } otherwise
000044    011211                             MOV    @R$2,@R$1    }    (make exchange;
000046    010312                             MOV    R$3,@R$2     }
000050    032700   000001                    BIT    #1,R$0       !is flag set?
000054    001403                             BEQ    L$11         !no - go decr. R2;
000056    062701   000002                    ADD    #2,R$1       !yes - incr. R1)
000062    000402                             BR     L$13
000064    062702   177776           L$11:    ADD    #-2,R$2
000070    005100                    L$13:    COM    R$0          !complement flag;
000072    000757                             BR     L$4          !go to beginning of loop;
000074    032700   000001           L$10:    BIT    #1,R$0       !is flag set?
000100    001403                             BEQ    L$15         !no - go incr. R1;
000102    062702   177776                    ADD    #-2,R$2      !yes - decr. R2;
000106    000751                             BR     L$4          !go to beginning of loop;
000110    062701   000002           L$15:    ADD    #2,R$1       !incr. R1;
000114    000746                             BR     L$4          go to beginning of loop;
000116    016646   000012           U$1:     MOV    12(SP),-(SP) }compute bounds for left
000122    010103                             MOV    R$1,R$3      }subfile; push them and
000124    062703   177776                    ADD    #-2,R$3      }recurse;
000130    010346                             MOV    R$3,-(SP)    }
000132    004767   177642                    JSR    PC,QSORT
000136    062701   000002                    ADD    #2,R$1       }compute bounds for right
000142    010116                             MOV    R$1,(SP)     }subfile; push them and
000144    016646   000014                    MOV    14(SP),-(SP) }recurse;
000150    004767   177624                    JSR    PC,QSORT
000154    062706   000006                    ADD    #6,SP
000160    012603                    L$2:     MOV    (SP)+,R$3    }pop stack and restore
000162    012602                             MOV    (SP)+,R$2    }registers;
000164    012601                             MOV    (SP)+,R$1    }
000166    000207                             RTS    PC           !return
Main program starts here:

                                    QUICKSORT:
000170    012706   001376                    MOV    #$STK,SP     !move stack limit to SP
000174    010046                             MOV    R$0,-(SP)    !push contents of R0
000176    010146                             MOV    R$1,-(SP)    !push contents of R1
000200    004767   177574                    JSR    PC,QSORT     !call QSORT
000204    000000                             HALT

                                    $QUICKSORT:


          000210                    SIGVAL: .=.+2
          000212                    SIGREG: .=.+2
                                    .GLOBL SIGVAL,SIGREG

          000212                    .ASECT
          000400                    .=400
          001400                    .=.+1000
          001376                    $STK=.-2
```

The Quicksort Program

Illustration 4.2

Contents of file DEMO1:

```
!DAME Routine DEMO1                                          Comments

DEMO1(LOAD('QSORT 20000)                          !load QSORT file starting at 20000

      LOAD('QSORT 30000)                          !load a second copy as data to be sorted

      IOBJ(R0 0 30000) IOBJ(R1 0 30100)           !insert bounds in registers

      LMR('DEFIO '*) EX(DEFIO)                    !load DAME file DEFIO; exec. routine DEFIO


      NTR()

      CR('C) IOBJ(C 0 0)                          !create C; initialize it; default node definition


      RUN(20170))                                 !start to run from 20170, QUICKSORT
!DAME Routine TIO
TIO(IF(C 'LT 5 (XX(CRLF) TI('NODE=)               !if C < 5 then
                 TIOBJ(CURNOBJ)                   !(type msg and current node-object

                 TI('INPUT) TI('=SET=)            !type curr. input set

                 TIOBJ(CISP)

                 TI('OUTPT) TI('=SET=)            !type curr. output set
                 TIOBJ(COSP)

                 BA('+ C C 1) )                   !incr. C)

                (DISAB(HTIO))))                   !otherwise disable hook
```

Contents of file DEFIO:

```
!DAME routine DEFIO - causes initialization, building
!and closing of input/output sets
DEFIO(HOOK('NE (IOS())  'HIOS)                    !initialize output set at node entry
     HOOK('OS (BOS()) 'HBOS)                      !build output set at each store operation
     HOOK('NX (COS())  'HCOS)                     !close output set at node exit
     HOOK('NE (IIS()) 'HIIS)                      !initialize input set
     HOOK('OF (BIS()) 'HBIS)                      !build input set at each fetch operation
     HOOK('NX (CIS()) 'HCIS)                      !close input set at node exit

     DISAB(HBOS)                                  !initially disable "build I/O set" hooks
     DISAB(HBIS)

     HOOK('NE ENAB 'HENB)                         !at node entry, enable them
     HOOK('NX DISAB 'HDISB))                      !at node exit, disable them again - in case nodes
                                                  !don't cover entire program
ENAB(ENAB(HBOS) ENAB(HBIS))
DISAB(DISAB(HBOS) DISAB(HBIS))
```

Illustration 4.3

```
*RUN DAME

 DAME11/10...

**LMR('DEMO1 '*)

**EX(DEMO1)
---FILE LOADED 20000  TO  20206
---FILE LOADED 30000  TO  30206          !Typed out by the LOAD commands in DEMO1

NODE:20170,,20200/4,,1/202057,,202055
INPUT-SET:(177707,,20172/20172,,1376/177706,,0/177700,,30000/1374,,0/177
7 01,,30100/1372,,0/20202,,177574/0/0 )
OUTPT-SET:(177707,,20000/177776,,0/177706,,1370/1374,,30000/1372,,30100/
0/0/0/0/0 )

NODE:20000,,20024/10,,1/205504,,205502
INPUT-SET:(177701,,30100/177706,,1370/1366,,0/177702,,0/1364,,0/177703,,
0/1362,,0/177707,,20010/20010,,1/177700,,30000 -->20014,,12/1374,,30000/
20020,,10/1372,,30100/0/0/0/0/0/0 )
OUTPT-SET:(177706,,1362/177776,,11/1366,,30100/1364,,0/1362,,0/177707,,2
0032/177700,,1/177701,,30000/177702,,30100/0 )

NODE:20032,,20034/2,,1/205375,,205373
INPUT-SET:(177702,,30100/177701,,30000/0/0/0/0/0/0/0/0 )
OUTPT-SET:(177776,,0/177701,,30000/0/0/0/0/0/0/0/0 )

NODE:20036,,20040/2,,1/205306,,205304
INPUT-SET:(177701,,30000/30000,,10146/177702,,30100/30100,,1403/0/0/0/0/
0/0 )
OUTPT-SET:(177776,,0/30100,,1403/0/0/0/0/0/0/0/0 )

NODE:20042,,20054/5,,1/205217,,205215
INPUT-SET:(177701,,30000/30000,,10146/177703,,0/177702,,30100/30100,,140
3/177707,,20052/20052,,1/177700,,1/0/0 )
OUTPT-SET:(177776,,0/177703,,10146/30000,,1403/30100,,10146/177707,,2005
6 /177700,,1/0/0/0/0 )
---HALT AT  20206
```

Let us now, for example, display the values of register 1 at the entry or exit from the most recent 3 node instances in whose I/O sets it appears.

```
**PLAYB(177701 3)
```

The format of the type-out is:

Instance Index		Starting Addr.	Instr.count at entry	Input set	Output Addr.		Value in I/O set
362:	NODE INST.	20160000000	4107	301516301542	4	OUTPUT VALUE :	30100
362:	NODE INST.	20160000000	4107	301516301542	4	INPUT VALUE :	30042
361:	NODE INST.	20154000002	4101	301566301612	6	OUTPUT VALUE :	30042

Example 2. Construction of Node Transition Matrix

A common type of model used for representing control flow is the so-called "transition matrix" M whose element M(i,j) is the number of transitions from node i to node j. By dividing each element in this matrix by the sum of all the elements in the same row, one can obtain a Markovian transition probability matrix P whose element P(i,j) is the probability of the next node to be executed being node j given that the current node is i. In this example, I shall give a DAME procedure for constructing the matrix M. Since we do not know the number of nodes, we cannot initially allocate the space for M. Thus, the approach we will take is to use the data in the NODETRACE table, which is maintained by DAME as the -ll program runs, to count the transitions between nodes. However, this table is of fixed size and when it is full, it is to be dumped onto disk and its pointer NTRACEPTR is set to zero. This pointer is initially 4, since the table initially contains some additional information in the first four words. We shall maintain the integrity of the matrix M by updating M each time the table NODETRACE is full, prior to dumping the latter onto disk.

To load the program and initialize the main memory, we shall use the DEMO1 routine of Example 1 except that the TIOBJ instruction for typing out I/O sets will be removed.

In the file named FLW, given in the next illustration, there are four routines: FLW, CHFLW, SFLW and FINDI. FLW is to be executed only when the table NODETRACE has been dumped onto disk for the first time.

FLW determines the number of existing nodes by taking the cardinality of the "node subclass master list" pointed by MNODESC; stores that value in E, sets $F=E^2$, declares the node transition table M as containing F words and the vector NV, which will contain the starting address of each node, as containing E words. It also creates the object G, which will be used to index into NV, and sets it to zero. It then searches the node subclass master list to determine the address of each node and fills in the vector NV. The node whose index in NV is i will be represented by the column i and row i and M.

The routine CHFLW is activated prior to each dump via the hook HNX1 inserted at run-time; it goes through the table NODETRACE and passes each node-address in chronological order to routine SFLW. SFLW computes the index into the table M for each node by calling FINDI to get the index into the vector NV of the node address passed to it, and updates M.

The success of this procedure clearly depends on the execution of every node defined in program at least once until the first dump so that FLW will see its name in the NODETRACE table and put its address in the node vector NV. However, if some node does not appear in NODETRACE, this will be detected by FINDI since that node will not be found in NV, and it will type the message 'EPROR-IN-FINDI' and return the control to the user. So, this procedure for constructing M is not foolproof, but it is efficient since it requires little monitoring activity between dumps of NODETRACE.

Illustrations 4.4 and 4.5 show the DAME routines and the protocol, respectively.

Illustration 4.3

```
!Routine FLW

FLW(CR('E) CR('F) CR('IND) CR('INC) CR('F1) CR('F2) :create some objects for later use
    CR('G) CR('M) CR('Q) CR('K) CR('J) CR('P) CR('OLDND)
    CR('L) CR('U) CR('I) CR('CNODE)

    IOBJ(L 0 4) IOBJ(U 0 3720)        :initial search limits for NODETRACE

    EVAL(E CARDINAL ', MNODESC)       :get no. of nodes, E

    BA('* F E E)                      :size of M, F = E^2

    CR('M 100 0 F) CR('NV 100 0 E)    :create M and NV; (ignore 100 and 0)

    IOBJ(G 0 0)                       :G ← 0

    SLST(M MNODESC (POP(M)            :Search List (SLST) works just like FISET; see 5.5.1

                   EVAL(M SEETHROUGH ', M)
                   IIOBJ(M M)         } get word Ø of node-obj. into M

                   BA('/ M M 1000000)      :get left half

                   IOBJ(NV G M) BA('+ G G 1) PUSH(0)))) :NV(G) ← M; G ← G+1; continue search
!Routine CHFLW
CHFLW(INCR(I L U 4 (BA('| J NODETRA I)      :get left half of every 4 word of NODETRACE into J
                   BA('/ J J 1000000)

                   PUSH(J)                 :pass if SFLW

                   EX(SFLW))))
!Routine SFLW
SFLW(POP(CNODE)                   :pop passed parameter into CNODE

      IF(OLDND 'EQ 0 (IOBJ(OLDND 0 CNODE) RET(2))) :if oldnd=0 then (oldnd ← cnode; return)

      PUSH(OLDND) EX(FINDI) POP(F1)                :F1 ← index of OLDND in NV

      PUSH(CNODE) EX(FINDI) POP(F2)                :F2 ← index of CNODE in NV

      BA('* IND F1 E)             :compute index into M

      BA('+ IND IND F2)

      BA('| INC M IND)             :get old count in M

      BA('+ INC INC 1) IOBJ(M IND INC)  :increment and store it back

      IOBJ(OLDND 0 CNODE))              :oldnd ← cnode
!Routine FINDI
FINDI(POP(Q)

      INCR(J 0 E 1                      :look for passed address in vector NV

            (BA('| K NV J)

            IF(K 'EQ Q (PUSH(J) RET(3)))))      :if found, return its index

      TI('ERROR) TI('-IN-F) TI('INDII) STOP())  :otherwise report error and stop
```

Illustration 4.5

```
.RUN DAME

 DAME11/10...

••LMR('DEMO1 '•) LMR('FLW '•)  !Load DAME files DEMO1 and FLW.  The hook HTIO has
                               !been removed from DEMO1
••HOOK('NX (IF(NTRACEP 'EQ 0 CHFLW)) 'HNX1) !hook HNX1 to be activated later

••DISAB(HNX1)                                  !disable it

••HOOK('NX (IF(NTRACEP 'EQ 0$
•••              (EX(FLW) EX(CHFLW) IOBJ(L 0 0)$   } will be executed after 1. dump only
•••               ENAB(HNX1) DISAB(HNX2))))$       } HNX1 will be activated after
•••               'HNX2)                           } subsequent dumps


!Having placed hooks HNX1 and HNX2 we can start up the program via DEMO1 which loads two copies of it, one
!to be used as data, and runs it.

••EX(DEMO1)
---FILE LOADED 20000  TO  20206
---FILE LOADED 30000  TO  30206
---HALT AT  20206                              !execution finished
••TOBJ(NV)                                     !type the node-vector NV
20170/20000/20032/20036/20042/20056/20070/20064/20074/20102/20110/20116/
20026/20160/20136/20154

••TOBJ(F)                                      !type size of matrix M
400

••IOBJ(J 0 0)

••INCR(G 0 377 1$   !type M as a 20x20 table.  $ is a continuation char.
•••      (BA('! 0 M G)$                        !Q ← M(G)
•••       TYIO(Q)$                             !type Q
•••       BA('+ J J 1)$                        !J ← J+1
•••       IF(J 'EQ 20 (XX(CRLF)$               !if J=20 then (start new line; J ← 0)
•••                    IOBJ(J 0 0)))))
0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0 13  0  0  0  0  0  0  0  0  0 12  0  0  0
0  0  0 144  0  0  0  0  0  0  0 12  0  0  0  0
0  0  0  0 43  0  0  0 100  0  0  0  0  0  0  0
0  0  0  0  0 25  0 16  0  0  0  0  0  0  0  0
0  0  0  0  0  0 25  0  0  0  0  0  0  0  0  0
0  0 25  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0 16  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0 52 26  0  0  0  0  0
0  0 52  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0 26  0  0  0  0  0  0  0  0  0  0  0  0  0
0 12  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0 12  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  7  6
0 12  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  3  3  0

    In the table typed above, element (i,j) is
the number of transitions from node i to node j
where i and j are the indices of node addresses in
vector NV.
```

Example 3. Analysis of Parallelism in the QUICKSORT Program

In this example, we shall consider the following problem: Suppose we have an unlimited number of processors connected in such a way that any processor is able to locate one idle processor at any time, pass to it a vector to be sorted and invoke it to execute in parallel with itself, thus enabling us to execute the first recursive call on QSORT in parallel with the second. Ignoring the instructions outside the main loop of QSORT, i.e., 20032 to 20114, estimate the elapsed time (as number of instructions executed) to sort a given vector.

Thus, the flow of control looks like a binary tree where each branch indicates one execution of QSORT. The time to be computed corresponds to the longest path in this tree. The DAME procedure given in Illustration 4.6 solves this problem by constructing a nested list structure representing the tree. Each list has two elements. The first element has two words containing the number of instructions which had been executed up to the entry and up to the exit from the node along that particular branch of the tree. The second element of each list is the sub-list representing the sub-tree under that node (Note: Here, I am using the word "node" in its usual meaning in DAME.).

Having created this DAME routine file beforehand, we can proceed to load and apply it as shown in Illustration 4.7. The tree represented by the nested listed structure, which has been typed out, is re-drawn in Illustration 4.8.

Each node in the tree shows the instructions executed along that branch up to the entry and the exit from the node as x/y, where x is the former and y the latter. y=0 indicates terminal nodes.

It can be seen that in the two longest paths through the tree, 536 instructions would be executed by the main-loop portion of the QSORT routine.

Illustration 4.6

```
!Routine QPAR
QPAR(LOAD('QSORT 20000) LOAD('QSORT 30000)   !load program
     IOBJ(R0 0 30000) IOBJ(R1 0 30040)       !set bounds for 20 elements
     CR('TEMP) CR('OBJAD) CR('ICT) CR('L1) CR('L2)
     CR('FLAG) CR('TEMP2)
     CR('ROOT) EVAL(ROOT MAKELIST) IOBJ(OBJAD 0 ROOT)  !create root of list, put it in ROOT
     PUSH(0)     !push initial amount of time passed=0          and OBJAD
     PUSH(OBJAD)!push address of list root

     HOOK('AIF PAR0 20000 20000 !MPAR0)
     HOOK('AIF PAR1 20032 20032 !MPAR1)
     HOOK('AIF PAR2 20116 20116 !MPAR2)
     HOOK('AIF PAR3 20136 20136 !MPAR3)
     RUN(20170))
!Routine PAR0
PAR0(IOBJ(FLAG 0 1)    !set flag to indicate new entry

     POP(OBJAD)        !get address of current object

     POP(TEMP)         !get amount of time already elapsed in this branch

     EVAL(L2 CREOBJ 100 0 2 0)  !create a 2-word object.  Put its address in L2

     IAOBJ(L2 0 TEMP)           !insert in its word 0 time already elapsed

     XX(INCLUDE ', L2 ', OBJAD));execute external -10 routine INCLUDE to put the new object in
                                 the current list
!Routine PAR1
PAR1(IF(FLAG 'EQ 1 (IOBJ(ICT 0 ICOUNT) IOBJ(FLAG 0 0)))) !if flag is set, get number of in-
!Routine PAR2                                              atructions through this node
PAR2(BA('- TEMP2 ICOUNT ICT)
     BA('+ TEMP2 TEMP2 TEMP)  }temp2 ← temp+icount - ict
     PUSH(TEMP2)              !pass it to PAR0 or PAR3

     IAOBJ(L2 1 TEMP2)        !insert total instructions in word 1 of new object

     PUSH(OBJAD)              !pass address of current list to ENDR

     PUSH(TEMP2)              !pass total instructions to ENDR

     EX(ENDR))
!Routine ENDR
ENDR(EVAL(L1 MAKELIST)        !create a new list.  Put its address in L1.

     EVAL(TEMP MAKEREP ', L1)
                               }include the new list as a member of current list
     XX(INCLUDE ', TEMP ', OBJAD)

     PUSH(L1) )                !pass address of new list to PAR0 or PAR3
!Routine PAR3
PAR3(POP(OBJAD) POP(TEMP) PUSH(TEMP) EX(ENDR)) !get values of OBJAD and TEMP; call ENDR
```

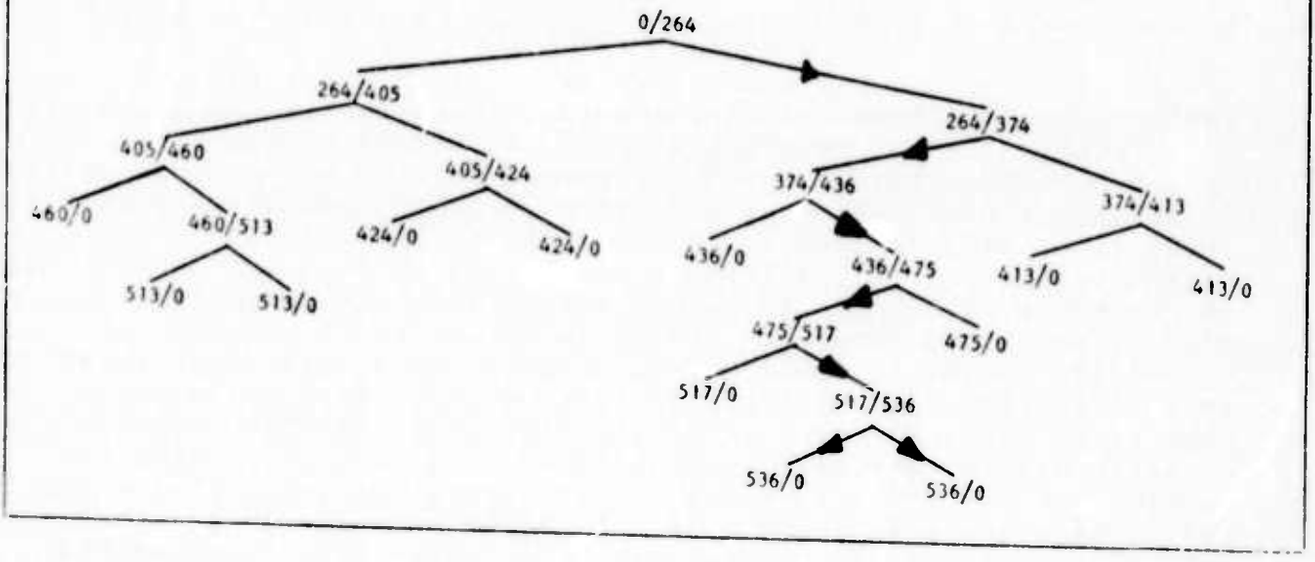
Illustration 4.7

```
•RUN DAME

 DAME11/10...

••LMR('QPAR '•)                          !load and execute QPAR

••EA(QPAR)
---FILE LOADED 20000  TO  20206
---FILE LOADED 30000  TO  30206
---HALT AT  20206                        !execution finished
••!!OBJ(ROOT)                            !type the list pointed by ROOT
[0/264 -->[264/405 -->[405/460 -->[460/0 ]-->[460/513 -->[513/0 ]-->[513
0/ ]]]-->[405/424 -->[424/0 ]-->[424/0 ]]]-->[264/374 -->[374/436 -->[43
6 /0 ]-->[436/475 -->[475/517 -->[517/0 ]-->[517/536 -->[536/0 ]-->[536/0
 ]]]-->[475/0 ]]]-->[374/413 -->[413/0 ]-->[413/0 ]]]]
```

The tree represented by this list is re-drawn below:

Illustration 4.8

Execution Tree Constructed by QPAR



Each node in the tree shows the instructions executed along that branch up to the entry and the exit from the node as x/y, where x is the former and y the latter. y=0 indicates terminal nodes.

It can be seen that in the two longest paths through the tree, 536 instructions would be executed by the main-loop portion of the given program.

Example 4. Data Flow Between Two Nodes

One class of analysis tasks for which DAME is most suitable is the determination of the data flow between two nodes, by finding the addresses which are both in the output set of one and the input set of the other. This analysis can be made in one or both directions of flow. If the node instances happen to be consecutive, then this procedure will yield the exact nature of the information passed from one to the other. If, on the other hand, there are intervening node instances, then one has to monitor their effect on the data flow between the two nodes. For a more detailed discussion of this question, see Chapter 2, Section 2.2.2.

Representation of the data flow from a node N to a node M for this purpose can, at the simplest level, be a list X of sets $X_i(N,M)$, $i=1,\ldots,k$, each set consisting of triples (A,B,C) where A is an address which is in the input set (strictly speaking, which is the "address part" of some member of the input set) of the i^{th} instance N_i of N and also in the output set of some instance M_j of M which occurred chronologically between N_{i-1} and N_i. If there are several such instances of M, then M_j is the latest one. In the above triple, B and C are the contents of A upon entry into N_i and upon exit from M_j, respectively. In this example, a DAME procedure DFLOW and the associated subroutines BUILD and COPY implement this process. These routines are given in the illustration below. The -11 program on which we operate is again the by-now-familiar QSORT. For the purposes of this example, we shall consider the data flow from node NA to node NB, extending from 20000 (i.e. relative address 0) to 20024 and from 20042 to 20072 respectively. NA is the first node in QSORT. It saves the contents of registers R1, R2 and R3 on the stack, gets the parameters (which are the lower and upper bounds of the vector to be sorted) into R1 and R2, checks to see if lower bound is less than upper bound and, if so, branches to the main portion which does the sorting. The second node, NB, is entered when two elements which have to be interchanged have been found. These elements are pointed by registers R1 and R2. NB makes the exchange, tests the flag which indicates which end of the vector is to be advanced in accordance with the quicksort algorithm, makes the advancement, complements the flags and branches back to the beginning of the loop. We note that while NA and NB are not consecutive, the intervening instructions in locations 20032 to 20040 do not modify any locations except PS and PC. As will be seen, neither of these appear in the input set of NB.

The DAME routines given here are quite general and could be applied to any two nodes by changing the limits of the nodes in the NODE and HOOK instructions in DFLOW and the FOSET instruction in COPY.

To explain briefly the functioning of the procedure, the main routine DFLOW does the following:

(i) Creates some objects which will be used later; of these, MAINL will point to the list we are interested in,

(ii) Creates a list and stores its address in MAINL,

(iii) Loads two copies of QSORT, one of which will be used as data,

(iv) Defines nodes NA and NB,

(v) Loads and executes the DAME routine DEFIO (this routine, as will be recalled from an earlier example, simply places hooks at node entry/exit points to build input/output sets),

(vi) Inserts a hook to execute routine BUILD after each execution of the instruction at location 20072, i.e. after NB,

(vii) Initializes registers R0 and R1 to contain the initial bounds of the vector to be sorted (which is what the main program QUICKSORT expects),

(viii) Starts the execution from location 20170, the starting address of QUICKSORT.

The routine BUILD is thus activated after each instance of NB and does the following:

(i) Creates a list, pointed by L,

(ii) Searches the current input set, pointed by CISP, for addresses which also occur in the output sets of the instances of NA and makes an entry in the list pointed by L for each such address.

(It will be recalled that an input set is a list of one or more ten-word objects, each word of which contains an address in the left half and the contents of that address in the right half. A zero indicates the end of the list. The address 0 is represented by 777777. C1 contains the address of current word to be looked at. The instruction Insert Indirect, IIOBJ(C2 C1), inserts in C2 the contents of the word pointed by C1. The COPY routine extracts

the address part of the word and uses the FOSET and FVAL instructions to find the most recent instance of the node NA in whose output set the address appears. If one is found, C6 will contain the contents of that address in the output set found; otherwise it will contain -1. (See the description of the FOSET instruction in Chapter 3, Section 3.6.2, for a better understanding of how this works.) If such an output set has been found, an object of 2 user words is created, pointed by C7. The contents of the word in the input set, and the value just found in the output set are inserted in it via the Insert Addressed (IAOBJ) instructions. The object is put in the list pointed by L, and COPY returns to BUILD. BUILD continues with the search until the current input set is exhausted. At the end of the SLST instruction, L points to a list of 2-word objects whose first word contains the data described above.)

(iii) Puts L in the main list pointed by MAINL and types out the list pointed by L.

Illustration 4.9

```
!routine DFLOW
DFLOW(CR('L) CR('C) CR('C1) CR('C2)
       CR('C3) CR('C4) CR('C5) CR('C6) CR('C7)
       CR('L1) CR('MAINL)
       !create a list; put its address in MAINL
       EVAL(MAINL MAKELIST)
       !load two copies of OSORT
       LOAD('OSORT 20000) LOAD('OSORT 30000)
       !define the nodes NA and NB
       NODE('NA 20000 20024) NODE('NB 20042 20072)
       !load monitor routine DEFIO and execute it
       LMR('DEFIO '*) EX(DEFIO)
       !insert hook to build desired lists
       HOOK('AIC BUILD 20072 20072
       !initialize R0 and R1, and go...
       IOBJ(R0 0 30000) IOBJ(R1 0 30040) RUN(20170))

!routine BUILD
BUILD(EVAL(L MAKELIST) !create a list for this instance; point L to it
!search current input list pointed by CISP
       SLST(C CISP (POP(C1) !get addr. of first 10-word object
                    !search each word
                    INCR(I 0 11 1 (IIOBJ(C2 C1) !get contents into C2
                                   IF(C2 'EQ 0 (RET(2))) !if 0, end search
                                   BA('+ C1 C1 1) !else, incr. C1
                                   EX(COPY)  )) !call COPY
                    PUSH(0))) !continue search

       XX(INCLUDE L '. MAINL) !put list pointer into main-list
       TIOBJ(L))      !type list for this instance

!routine COPY
COPY(BA('/ C3 C2 1000000) !get left half into C3
       !search output sets of node NA starting with most recent instance
       FOSET(C4 20000 (POP(C5) !get address of first object
                       FVAL(C6 C3 C5) !get value of address which
                                      !is in C3 from the object
                                      !pointed by C5 into C6
                       !if C6 <0, continue search else quit
                       IF(C6 'LT 0 (PUSH(0)) (PUSH(1)))))

!if search failed, exit routine
IF(C4 'LT 0 (RET(2)))
!otherwise, create a 2-word object; point C7 to it
EVAL(C7 CREOBJ 100 0 2 0)
!insert contents of C2 into word 0 of new object
IAOBJ(C7 0 C2)
!insert contents of C6 into word 1 of new object
IAOBJ(C7 1 C6)
!put new object in the list for the current instance
XX(INCLUDE '. C7 '. L))
```

Illustration 4.10

```
.LOG
JOB 14 CMU10A 7.04/DEC 5.04B TTY15
#C410BA07
PASSWORD:
1247     14-JUN-73        THUR
TH 1200...NEWS(6-14)

.RUN NDAME

DAME11/10...

**LMR('DFLOW '*) EX(DFLOW)
---FILE LOADED 20000 TO 20206
---FILE LOADED 30000 TO 30206
[177701,,30000/30000 30000,,10146/10146 177703,,0/0 177702,,30040/30040
30040,,3415/3415 177707,,20052/20052 20052,,1/1 177700,,1/1 20060,,2/2]
[17701,,30002/30002 30002,,10246/10246 177703,,10146/10146 177702,,
30040/30040 30040,,10146/10146 177707,,20052/20052,,1/1 177700,,177776
/177776 20066,,177776/177776]
[177701,,30002/30002 30002,,10146/10146 177703,,10246/10246 177702,,
300034/30034 30034,,3430/3430 177707,,20052/20052 20052,,1/1 177700,,
20060,,2/2]
[177701,,30004/30004 30004,,10346/10346 177703,,10146/10146 177702,,
30034/30034 30034,,10146/10146 177707,,20052/20052 20052,,1/1 177700,,
177776/177776 20066,,177776/177776]
[177701,,30004/30004 30004,,10146/10146 177703,,10346/10346 177702,,
30030/30030 30030,,453/453 177707,,20052/20052 20052,,1/1 177700,,1/1
20060,,2/2]
[177701,,30006/30006 30006,,12700/12700 177703,,10146/10146 177702,,
30030/30030 30030,,10146/10146 177707,,20052/20052 20052,,1/1 177700,,
177776/177776 20066,,177776/177776]
[177701,,30006/30006 30006,,10146/10146 177703,,12700/12700 177702,
30026/30026 30026,,5000/5000 177707,,20052/20052 20052,,1/1 177700,,
1/1 20060,,2/2]
^C
.
```

Example 5. Analysis of Instruction Mix and Addressing Mode Usage by PDP-11 Programs

This example is based on an experiment in which we were interested in comparing the performances of the PDP-11/20, /40 and /45 in connection with a proposal for the acquisition of several processors for the Carnegie Multi-miniprocessor (C.mmp). What we wanted was a rough estimate of the relative speeds with which these processors would execute programs typical of the workload to be placed on them here. The procedure followed was as follows:

(i) Four available programs were selected as benchmark programs: Two hand-coded in assembly language, and two BLISS/11 programs. The assembly language programs were an interactive disassembler for the PDP-11 written by Roy Levin and the "vector mode" portion of the XGP (Xerox Graphic Pointer) support program written by George Robertson and Hal Van Zoeren. The two BLISS/11 programs were an interactive PDP-11 debugging aid written by Roy Levin and the Quicksort program used in the preceding examples. These programs were judged to be a good cross-section of the workload to be run on the C.mmp, excluding the number-crunching programs.

(ii) The information required to project the performances of the models /40 and /45 were derived from the respective processor manuals,

(iii) A DAME routine (IMIX) was written to monitor the execution of these four programs and gather the required data,

(iv) A DAME routine (RPORT) was written to summarize and report the collected data in the form of instruction mix, addressing mode usage and branching statistics,

(v) Two BLISS/10 programs were written to calculate the performances of each of the /40 and /45 (These were needed because of the wide dissimilarity in the forms of the processor specifications given in the manuals for the two machines.) These programs were written in BLISS rather than DAME because of the relatively large amount of arithmetic, table-look up etc. that was required. This fact also turned out to be a good test of the ease with which data could be communicated between DAME and BLISS, which was found to be very easy and natural.

(vii) The DAME routines and the BLISS models of the /40 and /45 were debugged and hand-checked over short sequences of -11 code,

(viii) Several runs of varying lengths were made with each of the four benchmark programs with different inputs. The collected data was incorporated into a memorandum and sent to various faculty and staff members connected with the C.mmp project.

In this example, I shall go over the IMIX program. As mentioned above, the function of this routine was to build various tables and accumulate counts during execution. Below is a list of these data items (all integers below are decimal; in the listing of the IMIX routine itself in the next illustration, in octal):

DOTAB: a 12x8 table containing a count of each of the twelve double-operand instructions broken down by the eight destination modes,

SOTAB: a 26x8 table for single-operand instructions, format similar to DOTAB,

TOTICOUNT: a vector containing a count for each op-code (indexed by OPN- see below),

DOSNO: a 12x8 table for double-operand instructions whose source operand mode > 0, broken down by destination mode,

DOSO: a 12x8 count table for double-operand instructions whose source operand mode= 0,

TOTSMOD: a 12x8 count table for double-operand instructions by source mode,

JSRCO: a count vector for JSR instructions by dst. mode,

DSR7: a count of instructions whose destination operand is register 7 (PC),

JMPR7: a count of JMP instructions whose destination operand is register 7,

TOTDO: total number of double-operand instructions,

TOTSO: total number of single-operand instructions,

TOTCCOC: total number of condition code operators,

TOTBP: total number of conditional branch instructions,

SUCCBR: total number of successful conditional branches,

BRPD: total distance covered by positive branches,

PBCNT: total number of positive branches,

BRND: total distance covered by negative branches,

NBCNT: total number of negative branches,

UNSUCCB: total number of unsuccessful conditional branches.

In performing these calculations, IMIX uses a number of data items supplied by the simulator. These are (all items refer to the current -11 instruction):

OPN: a unique integer representing the op-code,

(Note: the op-code itself is not suitable for this purpose)

SRCMODE: source operand mode,

DSTMODE: destination operand mode,

DSTREG: destination register,

OPC: character representation of mnemonic op-code,

OLDPC: last value of PC,

The IMIX routine itself is given in the next illustration. The protocol and the results of the analysis are not given here because that would require the inclusion of the RPORT routine as well and possibly also the BLISS routines for projecting the performances of the /40 and /45. I do not consider the actual results of that analysis as important for this thesis as the description of the methodology.

Illustration 4.11

```
IMIX(CR('C1 300 0 1) CR('TOT)
     CR('DOTAB 100 0 140) !will contain d.o. instr. counts by dst. mode

     CR('SMPCT 100 0 140) !table for src. mode percentages

     CR('SOPCT 100 0 320) !table for single opd. instr. percentages

     CR('TOTDO) CR('TOTSO) CR('TOTBR) CR('TOTMS)
     CR('JSRCO 100 0 10)
     CR('BRPD) CR('BRND)
     CR('PBCNT) CR('NBCNT)
     CR('T1) CR('T2) CR('T3) CR('T4)

     !insert hook to execute MIX after every instruction
     HOOK('IC MIX 'HMIX))

MIX(BA('! T1 TOTICOU OPN) !increment TOTICOUNT[OPN]
    BA('+ T1 T1 1)
    IOBJ(TOTICOU OPN T1)

    !decode OPN, call for appropriate action
    IF(OPN 'LE 13 INDO !if double operand, call INDO
         (IF(OPN 'LE 45 INSO !if single operand, call INSO
               (IF(OPN 'LE 57 INCOP !if cond. code opr., call INCOP
               !if conditional branch then call INCBR,
                    !else incr. "misc. instruction" count
                     (IF(OPN 'LE 77 INCBR (BA('+ TOTMS TOTMS 1)
                                           !if JSR, call INJSR
                                           IF(OPN 'EQ 100 INJSR)))))))))

!double-operand instruction handler
INDO(BA('* T3 OPN 10) !compute index into TOTSMOD table, incr. table entry
     BA('+ T2 T3 SRCMODE)
     BA('! T1 TOTSMOD T2)
     BA('+ T1 T1 1)
     IOBJ(TOTSMOD T2 T1)
     !incr. DSR7 if required
     IF(DSTMODE 'EQ 0 (IF(DSTREG 'EQ 7 (BA('+ DSR7 DSR7 1)))))
     !increment count according to whether srcmode is 0 or not
     IF(SRCMODE 'GT 0 INCGO INCE0)
     !incr. total d.o. count
     BA('+ TOTDO TOTDO 1))

INCGO(BA('+ T4 T3 DSTMODE) !incr. count for d.o. instr. with srcmode>0
      BA('! T1 DOSN0 T4)
      BA('+ T1 T1 1)
      IOBJ(DOSN0 T4 T1))
```

(continued on next page)

Illustration 4.11
(continued)

```
INCEO(BA('+ T4 T3 DSTMODE) !incr. count for d.o. instr. with srcmode=0

      BA('! T1 DOSO T4)
      BA('+ T1 T1 1)
      IOBJ(DOSO T4 T1))

!single-operand instruction handler
INSO(BA('- T3 OPN 14) !compute index into SOTAB
     BA('* T3 T3 10)
     BA('+ T4 T3 DSTMODE)
     BA('! T1 SOTAB T4)
     BA('+ T1 T1 1)
     IOBJ(SOTAB T4 T1) !incr. SOTAB entry and store it back
     BA('+ TOTSO TOTSO 1) !incr. total s.o. count
     !increment DS7 and JMPR7 if required
     IF( DSTMODE 'EQ 0(IF(DSTREG 'EQ 7
                          (BA('+ DSR7 DSP7 1)
                           IF(OPC 'EQ 'JMP (BA('+ JMPR7 JMPR7 1))))))))

!increment total cond. code operator count
INCOP(BA('+ TOTCCOC TOTCCOC 1))

!increment total branch count, take care of successful and unsucc. branches
INCBR(BA('+ TOTBR TOTBR 1)
      BA('- T1 PC OLDPC)
      IF(T1 'NEQ 2 INCSB INCUB))

!successful branch
INCSB(BA('+ SUCCBR SUCCBR 1)
      !accumulate positive(forward) and negative(backward) branch
      !distances and counts
      IF(T1 'GT 0 (BA('+ BRPD BRPD T1) BA('+ PBCNT PBCNT 1))
                  (BA('+ BRND BRND T1) (BA('+ NBCNT NBCNT 1))))

!unsucc. branch
INCUB(BA('+ UNSUCCB UNSUCCB 1))

!increment JSR
INJSR(BA('! T1 JSRCO DSTMODE) !incr. JSRCOUNT-by-DSTMODE
      BA('+ T1 T1 1)
      IOBJ(JSRCO DSTMODE T1)
      IF(DSTMODE 'EQ 0
      !incr. JSRR7 if required
         (IF(DSTREG 'EQ 7
             (BA('+ JSRR7 JSRR7 1))))))
```

CHAPTER 5

A PERFORMANCE MODEL FOR DAME-LIKE SYSTEMS

Having given a description of the design of the DAME system and illustrative examples of its application in various types of analysis tasks, it is now worthwhile to consider the resource requirements and performance of DAME-like systems. It should be clear by now to those who have examined Chapters 3 and 4, that such systems are very costly in terms of main storage and CPU time. Thus in this chapter, I would like to construct a model of the operation of DAME-like systems and parameterize the resource requirements of each major component of that model. To do this, I shall proceed as follows: First, I shall give an informal and intuitive definition of what I mean by "DAME-like" systems (Section 5.1). Then, I shall construct a more structured and concise model of such systems, exhibiting the overall control flow structure and the main "cost centers" ignoring the costs incurred by any hooks, i.e. involving only the object machine simulator and checks for hooks (Section 5.2). This will be followed by a characterization of the overhead of two major types of monitoring operations which are essential to our approach; namely, the monitoring of node entry and exits (including the maintenance of the node trace table) and the construction of input/output sets (Section 5.3). These operations, while they are implemented in the DAME system by the insertion of hooks by the system itself just as a user would insert hooks, should be regarded as integral parts of the analysis facility and hence, their performance is considered a significant part of the basic performance of such a facility. Thus, at the end of Section 5.3, we will have constructed a rough theoretical model of the object machine simulator including the checks for every type of hook defined in the DAME system and we shall have superimposed on this model, a model of the overhead of the execution trace facility, i.e. the node and input/output set mechanisms. This will provide a picture of the overall operating overhead of such a facility excluding any user hooks. The amount of everhead introduced by user hooks is, of course, a function of the actions performed by the specific hooks and therefore cannot be modelled in general.

Finally, in Section 5.4, some measurements of the PDP-11 simulator, the node entry/exit overhead and the input/output set overhead are given.

5.1 An Informal Characterization of DAME-like Systems

We shall call a system "DAME-like" if its principal goal is the monitoring and dynamic analysis (as opposed to post-mortem analysis) of the behaviour of the object system by (i) permitting the user to define a structure over his program code, (ii) collecting execution history data in terms of the components of that structure in such a way that backtracking to any point in the execution history is possible, and (iii) permitting the user to perform arbitrary computations over the present state and the state history of the object system at every operand access, instruction fetch/completion and structural component (node) entry/exit. A limitation of DAME-like systems, as structured here, is that they operate on single-stream, sequential processors where the system state resulting from any sequence of instructions can be completely determined from the initial state of the processor and the inputs. This means, for example, that DAME-like systems are not well suited to analyzing the behaviour of programs with a heavy dependence on the timing of asynchronous I/O devices, since the latter in reality are parallel processors. Backtracking over periods of time in which such devices caused interrupts or turned on status bits in their device control registers are difficult to accomplish with DAME-like systems.

5.2 A Model of DAME-like Systems

The basic operational cycle of a DAME-like system consists of the instruction cycle of the object machine with the addition of checks for monitor hooks and the actions initiated by the hooks themselves. The operand addresses of an instruction are assumed to be decoded and computed sequentially. Side-effects arising from the operand decoding/computation process (e.g. auto-increment/decrement in the PDP-11) are also viewed as additional operand accesses and thus are subject to checks on such accesses. Arbitrary levels of indirect addressing will be permitted. On the other hand, for the purpose of keeping the exposition simple, only single-word operands (i.e. no block transfers, half word or byte-addressing) will be considered. Even with this restriction, it is impossible to give a single, accurate and constructive model of the instruction decoding process which will describe all conveivable processors satisfying this restriction. In the model given below, we assume the following kind of an instruction decoding process: The ith instruction is fetched; its opcode is determined; the number of operands is determined; the address of each operand is determined and each operand is fetched or stored, one at a time, according to its access-type as determined from the instruction; each store operation is usually preceded by a computation of the value to be stored from the operands which have been fetched so far.

We can break down the total cost, C, of the simulated execution of an object machine instruction into 3 parts:

1- The basic cost, C_B, of indexing into the object machine memory to get the instruction, executing the instruction and checking for interrupts,

2- The cost of scheduling memory access events and updating the clock (C_S),

3- The cost of checking for hooks at each contact point (C_H).

Clearly, these cost components are not incurred in lumps, but rather they are interleaved throughout the execution of each instruction. C_B depends on the semantics of each instruction and how easily it can be emulated on the host machine.

C_S is a direct function of the number of events to be scheduled. In a memory-cycle level simulator, for an instruction involving n operands in the main memory, $C_S = (n+2)T_S$ where T_S =the cost of scheduling an event and activating it. The two events in addition to the n memory accesses for operands, are for simulating the delays for fetching the instruction and performing the operation.

C_H is a direct function of the total number of operands fetched or stored, including side-effects, by the instruction. It involves two kinds of overhead: checking for general hooks and checking for addressed hooks. Thus, for an instruction involving a total of m operands, $C_H = (m+2)(T_{GH} + T_{AH})$, where T_{GH} = overhead of checking for general hooks, T_{AH} = overhead of checking for addressed hooks, and the two additional checks are for checks for instruction fetch and instruction completion hooks.

Thus for a simulator, which has been written in a "loose" way so that inserting checks for hooks will not cause much perturbation, if the average number of operands of an instruction which are located in the main memory is n, then the ratio R_1 = (simulation time/real time) with no checking for hooks will be

$$R_1 = (C_B + (n+2)T_S)/T_r$$

where T_r is the average time to execute the same kind of instruction (i.e. involving n main memory accesses) on the real object machine. If we add to this the overhead for checking for hooks with an average number, m, of total operands per instruction we get

$$R=(C_B+(n+2)T_S+(m+2)(T_{GH}+T_{AH}))/T_r$$

which is a broad-gauge, general model of the performance of a DAME-like system with no hooks attached. If the object machine simulator has been implemented at the instruction level, rather than at memory cycle level, then the associated overhead can be found by setting n=0. Further, if hooks can only be inserted at instruction fetch/completion level, rather than operand fetch/store level, the corresponding overhead can be found by setting m=0.

5.3 The Overhead of the Node Mechanism

The overhead introduced by the Node Mechanism can be considered in two parts: (i) the overhead due to checking for entry and exits from nodes, and (ii) the overhead for the construction of input/output sets. Let us consider these two components in turn.

5.3.1 The Overhead of Detecting Node Entry and Exits

Let us first consider the case where nested nodes are not permitted. In this case the procedures for detecting node entry and node exit, which I shall denote by ENTRYP and EXITP respectively, can help each other significantly by communicating to each other information as to whether an entry or exit has been performed. Since nested nodes are not permitted, each node entry must be followed by a node exit before another node entry can occur. Similarly, every node exit must be followed by a node entry before another node exit can occur. Further, since we do not assume that the defined nodes cover the entire program, there will be times when the control flow will not be inside any node. Hence, after EXITP tells ENTRYP that the last node has been exited and therefore that a new node may begin anytime, ENTRYP must check with each subsequent instruction fetch to see if a new node is being entered. The cost of this check will depend strongly on its implementation. For example, if there are two additional bits in the representation of the object machine available for this use, these can be used to indicate the first and the last instructions of a node. Otherwise, a list of node definitions can be searched; or alternately, as in DAME, each used memory location can be assigned an "attribute list" and a node descriptor can be put on the attribute list of

the starting address of the node. In the last alternative, one proceeds as follows after each instruction fetch:

1- See if current instruction address has an attribute list (most addresses won't);

2- If not, it can not be a node entry; hence, return;

3- See if there is a node-descriptor object on the attribute list;

4- If not, return;

5- Compare the node starting address given in the node descriptor object with the current instruction address to make sure they coincide.

The longest step in this procedure is step 3, and even that step is not very long since there usually aren't more than four or five items on the attribute list of any location. The real cost of this procedure lies in the inclusion of an attribute list pointer potentially for every object machine location.

If the chosen approach for detecting a node entry is searching the list of node definitions to see if there is a node starting at the current instruction address, then, assuming a binary search over a list of n node descriptors ordered by their starting addresses, the average number of comparisons will be on the order of $\log_2 n$.

Each of these three approaches for detecting a node entry requires the associated checking to be done with every instruction executed after the last node exit until a new entry is detected. Thus, the total overhead caused by any one of the three is also a function of the total number, Q, of such instructions executed. If we denote by S the ratio of the number of executed object machine instructions which belong to a node to the total number of object machine instructions executed, and by O_{NE} the overhead per instruction caused by the particular approach for node detection, then the average overhead per instruction caused by the ENTRYP procedure, without nested nodes, will be $O_{NE}(1-S)$. This formula means, for example, that if there are large segments of executed code which do not belong to a node, this may cause a significant overhead.

The procedure for detecting the exit from the current node, again assuming no nested nodes, is much simpler and requires a comparison operation after each instruction in the node. Thus, if we denote by O_{NX} the cost of making a comparison, the overhead for detecting the exit from a node is $O_{NX} S$.

In addition to detecting entry and exits, there is a cost for creating an entry in the Node Trace table for each node executed. Let us denote that overhead by O_{NT}. If the average number of instructions per node instance is I_{NI} then this overhead is $O_{NT} S/I_{NI}$.

5.3.2 The Overhead of I/O Set Maintenance

Now let us consider the largest component of cost associated with the Node Mechanism, namely the construction of input-sets and output sets.

The construction of an input-set involves the following general steps:

I1- At node entry, allocate space for the set,

I2- After every fetch operation, determine if the fetch address is already in the input-set or the output-set (i.e. if it has been fetched or written previously in this node instance),

I3- If not, add the address and its contents as an element to the input set.

The construction of an output-set similarly involves the following steps:

O1- At node entry, allocate space for the set,

O2- Before each store operation, determine if the store address is already in the output-set,

O3- If not, add the address (with an undefined content) as an element to the set,

O4- At exit from the node, fill in the current contents of all the addresses in the output-set.

Since, in general, the size of an I/O set can not be predicted in advance, some decision has to be made as to how space will be allocated. It clearly is wasteful to obtain new space for each element and link it to the rest. There are similar problems with completely static allocation. The best procedure seems to be some kind of a compromise between the two. (In DAME, this is handled by obtaining space in 10-word chunks, each word to contain an (address, value) pair. Unused words will contain -1 in the address half. These 10-word chunks are put in a list. The list-head has one user word which contains the index of the next slot in the last member of the list.) Let us denote by IS_L and IS_E the everage everhead for creating the list head and for adding a new element, respectively.

The cost of determining whether or not an address should be added to an I/O set (i.e. whether it is a new or an existing address) depends strongly on the implementation. In DAME, this is done by using bits 16 and 17 (from the right) of the PDP-10 word representing an -11 word, to indicate those words which are already members of the output set and the input set respectively. Hence, the overhead amounts to testing these bits of each word being accessed, and possibly setting one of them. If we denote by w, B_1 and B_2 the ratio of the number of distinct operands to the number of total operands, the overhead of testing a bit and the overhead of setting a bit, respectively, then the overhead of this approach for the input and output sets, per instruction inside a node, is $2m(B_1+wB_2)$ and per node, it is $2I_{NI}\, m(B_1+wB_2)$.

Let us now consider the case when the implementation does not permit this approach (i.e. there are no available bits). Let us suppose that the "brute force" method of searching the I/O set to determine if a given address is in it or not is being used. Whenever an address is generated, the average number of existing elements in an I/O set is $wmI_{NI}/2$, the number of comparisons caused by new elements is $w^2mI_{NI}/2$ and the number of comparisons caused by old elements is $(1-w)wmI_{NI}/4$. Thus, the average total number of comparisons for constructing the input and the output sets of a node instance using this approach, assuming that the above parameters are equal for both input and output sets, is:

$$2(w^2mI_{NI}/2+(1-w)wmI_{NI}/4)$$
$$=wmI_{NI}+(w^2mI_{NI}/2)$$

Then, the average overhead, O_{IO}, per executed object machine instruction for constructing I/O sets is:

$$O_{IO} = (S/I_{NI})*(wmI_{NI} + (w^2 mI_{NI}/2))$$

$$= Swm(1+w/2)$$

where S and I_{NI} are as before.

We are now in a position to give an estimate of the average total overhead, O_I, per executed object machine instruction:

$$O_I = C_B + (n+2)T_S + (m+2)(T_{GH} + T_{AH}) + O_{NE} + (1-S)+O_{NX} + S+O_{IO}$$

where

C_B = the average cost of emulating one object machine instruction, with no event scheduling or checking for monitor hooks,

n= average number of main memory accesses per OM instruction,

T_S = the cost of scheduling an event and activating it,

m= total number of operands per OM instruction,

T_{GH} = overhead of checking for a general hook,

T_{AH} = overhead of checking for an addressed hook,

O_{NE} = overhead per OM instruction of detecting a node entry,

S= ratio of OM instructions belonging to some node to the total number of executed OM instructions,

O_{NX} = overhead per OM instruction of detecting a node exit,

w= ratio of the number of distinct operands to total operands generated over the course of the execution,

O_{IO} = overhead per executed instruction due to construction of I/O sets.

5.4 Measurements of the DAME System

In this section, some measurements of the overhead of the DAME system along the lines outlined above will be presented. First, a disclaimer note is in order: as mentioned previously, the minimization of the resource requirements was not a primary goal in the design and implementation philosophy of the DAME system and often these goals were neglected in favor of flexibility in the analysis facilities offered in order that new and useful features may be discovered. This philosophy has, in this author's opinion, met its goals. On the other hand, the performance has been worse than expected. Thus, the real purpose of this section is to give the reader an idea of what to expect in the way of the "relative", rather than "absolute", performance of a DAME-like system in the various monitoring and analysis tasks on which measurements are presented. Clearly, the speed of any component can be increased by better coding or less generality or by the use of some of the ideas presented in the final section of Chapter 3.

5.4.1 Performance of the PDP-11 Simulator

The most basic observation is that simulation at memory cycle level via a general-purpose scheduling mechanism degrades the performance by at least a factor of 3 over emulation, in which no scheduling is made. In the DAME system, simulation runs about 3000 times slower and emulation 1000 times slower than a PDP-11/20. These factors include about a 25% overhead for checking for hooks. These figures are based on measurements of the time charged to the user by the PDP-10 monitor, which includes supervisory and swapping overhead etc. and have shown a deviation of up to 15% in both directions.

5.4.2 Node Entry/Exit Overhead

If input/output sets are not being constructed, the overhead for user-defined nodes amounts to 3.2 milliseconds per node instance for entry and exit combined and 1.2 milliseconds per node instance to create a node trace entry, for a total of 4.4 milliseconds per node instance. In the DAME system, these costs have been found to be only associated with the actual entry and exit events; the cost of checking for entry and exit with each instruction is found to be less than the precision of the measurements.

5.4.3 Input/Output Set Overhead

When I/O sets are being used, there is an added overhead at node entry and exit, of about 40 milliseconds each, for creating and closing the I/O sets. In addition, the overhead of testing each generated address to see if it should be added to the input or the output set amounts to about 1.3 millisecond per fetched or stored operand, or about 6 milliseconds per OM instruction in the node instance. Thus, for a node instance of 5 instructions, the total overhead for I/O set creation and maintenance would be (2*40)+(6*5)=110 milliseconds. If we assume that 40 percent of all the executed instructions belong to some node and an average of 5 instructions per node instance, the total overhead for nodes and I/O sets would be 11.5 ms per executed instruction.

For a PDP-11 simulator with a slow-up factor of 3000, assuming an average of 3.5 microseconds of real-time per instruction, this amounts to an additional delay factor of 1.8.

CHAPTER 6

HIGH-LEVEL LANGUAGES FOR EXECUTION ANALYSIS

One of major shortcomings of the DAME System as described in Chapter 3 is that its language is too primitive for making arithmetic calculations and certain types of monitoring operations. This fact was not altogether unexpected. One reason for choosing this level in the design was the desire to avoid interpreting by software a complex syntax at run-time. A second reason was the anticipation of the possibility that any proposed hardware or microcoded implementation of a DAME-like facility might employ an instruction set very similar to this one. Hence an effort was made to keep a major part of the instruction set simple enough to be implemented by hardware or, more probably, by microcode. However certain instructions are still too complex and would probably be best implemented by software (e.g. Playback Values, Replay Node Instance, Type Object instructions).

In this chapter, I would like to discuss some issues in the design of high-level languages for execution monitoring and analysis. The emphasis will be on features which are particularly relevant to this application area.

The general structure of this chapter is as follows: first, a number of issues related to the human engineering aspects of interactive systems and languages are discussed as they apply to our problem. In particular, trade-offs between simplicity and power and between terseness and "rememberability" (ease of use) are outlined.

Second, the major data elements with which a high-level execution analysis language must deal and the appropriate forms of access to each of these data elements are taken up.

Finally, the problem of "continuously-evaluated" expressions is discussed. In particular, appropriate control structures for the continuous evaluation of a set of predicates and techniques for efficient implementation, as discussed by D. Fisher in this thesis [Fi 70], are presented and evaluated.

6.1 Some Human Engineering Issues

Since most of the programming in the analysis level will be done by the analyst at the terminal, almost in real-time, without laboring over a page of analysis code for several hours, certain properties of the total interactive system become very crucial. The issues I would like to discuss here are those related to this aspect of the design of the language of the analysis facility.

Due to the hands-on, "quasi-real-time" nature of the analysis programming process, it is clear that the language must be terse and conducive to error-free programming. An error in an analysis procedure can be "doubly costly" in the sense that it not only causes a wrong computation but also unnecessary periods of (possibly simulated) execution by the object machine which it controls. Especially the control structure of the analysis language is an important factor, because of possible interaction between the control flow in the analysis program and the object program. Another complicating factor is that several analysis actions may have been independently scheduled to be activated at the same contact point. Hence, whenever these actions are sensitive to the order in which they are executed, the user must have explicit knowledge of that order and must be able to modify it. In a list-oriented system such as DAME, this is extremely easy. Here, the flexibility of a loosely structured list must be weighed against the execution efficiency of a more optimized, tighter representation.

We have already noted the need for terseness, simplicity of syntax and conduciveness to error-free programming. These objectives can conflict with each other when any one of them is pursued with excessive zeal. For example, the goal of terseness can lead to a design where the user has to remember a large number of special symbols as operators or control characters. Over-emphasis on simplicity of syntax can lead to either weakening of the power of the language (as one goes in the general direction of the Turing tar pit) or to the definition of many special symbols which have to be remembered. An interesting case is presented by the syntax of LISP. It neither requires memorizing a large number of special symbols, nor can the language said to be too primitive. Its failing however, as users of LISP will painfully testify, is the extreme reliance on balanced and properly matched sequences of parentheses, which is one of the most frequent sources of simple errors in LISP programming. Another virtue of LISP is the fact that both program text and data use the same basic representation: namely, list structures. This feature facilitates operations on programs as data, e.g. to parse them, generate them or delete them. These operations are more difficult in languages where the syntactic elements of the language can not

be represented in one of the dominant data types or data structures defined in the language (it must be noted that most primitive machine languages do satisfy this requirement). For these reasons, a list-oriented syntax was selected for DAME. While there is much room for improvement in it, the chosen syntax has proved remarkably flexible and resilient under demands to accomodate more and more complex instruction forms. A good example of this is the Search List(SLIST) instruction. (See Section 3.6.1 or Appendix A: Introduction to DAME for a description of this instruction).

I would now like to consider the special-purpose data structures with which high-level execution analysis languages must deal (i.e. structures unique to execution analysis) and the access methods which they must provide.

6.2 High-Level Data Access in Execution Analysis

The set of major data elements with which an execution analysis facility must deal were discussed in Chapter 2, and we summarize those elements here:

(i) The external state of the Object Machine (i.e. main memory and user-addressable registers),

(ii) Some parts of the current internal state of the OM,

(iii) Possibly, user program text and symbol table,

(iv) Structural information about the user program (e.g. its nodes),

(v) Empirical data associated with each component of the structure (e.g. I/O sets of node instances, data created by user at run-time),

(vi) Execution history,

(vii) Analysis program text,

(viii) Representation of the association between analysis actions and contact points,

(ix) Entities holding intermediate results of analysis computations.

I shall now discuss appropriate forms of high-level access to each of these elements.

(i) The external state elements should be accessible by explicit addressing, e.g. core[2000], by computed addresses,

e.g. core[A+B], through Object Machine pointers (e.g. core[A+ core[B]]), as well as in blocks (e.g. core[A:B] ← 0, where A:B denotes 'A to B', or core[100:200] ← core[300:400]). User-addressable registers should be accessible by their mnemonic names used in the assembly language as well as by their memory addresses where such addresses exist.

(ii) Those elements of the internal state of the object machine which contain the various fields of the current instruction (e.g. opcode, source operand, destination operation) should be accessible by suitable mnemonics.

(iii) Access to user program text makes possible such things as building a text editor/incremental assembler into the analysis facility so that corrections to user programs may be made as they are discovered, rather than saved until the end and made in a seperate operation. The availability of the user symbol table clearly facilitates communication between the user and the analysis facility by permitting the use of the symbols appearing in the user program. One or both of these facilities are available in several systems though not in DAME (e.g. See Lampson [La 65], Evans and Darley [ED 65]. For a comparative discussion of various techniques related to this topic, see Evans and Darley [ED 66].)

(iv) Structural information about the user program describes the components of that program, as they have been defined by the user or determined by the system for the purposes of analysis, and the relationships between the components (e.g. predecessor/ successor, outer/inner node relations). This can be in tabular form or in the form of "descriptor objects" (as in DAME) which can be manipulated by list-processing functions. In any case, it should be possible to reference these descriptions explicitly (e.g. "node A" or "the node starting at location 10000"), by a computed address (e.g. "the node starting at the location pointed by contents of location 10000 + contents of register 3"), or as elements of a list or table satisfying a predicate (e.g. "all nodes between 10000 and 12000"). Better yet, the user can be given a facility for stepping through the component descriptions in a systematic way and computing arbitrary functions using the various fields within each description, with the ability to exit the search at any point or have it terminated automatically when the end of the table or list is reached.

(v) Empirical data generated during the execution of the user program should be linked with the phase of the execution to which they relate and they should be accessible by the user through that link. An example of such data is the input/output sets of a node instance in DAME. These sets are accessible directly via pointers contained in the entry for the associated node instance in the node trace table, as well as through the chrono-

logical list of pointers to a node's I/O sets, pointed by the node object itself. This process of linking empirical data with the associated portion of the execution history can be done by the Analysis Facility for specific types of data which the system knows about (e.g. I/O sets); but the user should also have a way of doing the same thing for arbitrary data. An example of the latter case is where the user wants to attach to each node a list of the addresses of every unique successor of that node in an analysis of control flow. This requires, for example, that whenever a new node is entered, the user be able to locate and search the current members of the successor list of the last node for the address of the new one, and if it is not found, be able to add it there. Such a mechanism may be implemented in terms of a more general associative search facility, such as LEAP in SAIL (See Feldman and Rovner [FR 69]). This associative search facility would permit LEAP-like statements such as:

```
FOREACH X,Y,Z SUCH THAT <condition> AND
              <condition> ...  DO <statement>;
```

where X, Y and Z may be nodes, node instances, I/O sets of any of the other defined object types in the system. (Note that one would probably prefer terse, single-character symbols for the FOREACH, SUCH THAT and AND in an interactive language).

(vi) The execution history information represents essentially a variable-level trace, where the precise level depends on the structure which has been defined over the user program. Thus, the time grain and the volume of collected data is under user control. In addition to the normal maintenance by the Analysis Facility over the course of the execution, this data will also be accessed by user analysis routines. The form of the access should be similar to the preceding case; e.g. applying a function f to a sequence of elements in the execution history which satisfy a user predicate p. The function f and the predicate p may involve both the history information itself (e.g. the address of the k nodes executed prior to the last execution of node A) or the empirical data dynamically associated with it, as described in (v). For this purpose, the analysis language should have a facility for searching over the execution history events, backward or forward in time, and applying predicates to each event encountered.

In the last two paragraphs, two distinct ways for performing searches over execution history and associated empirical data have been proposed. It is beneficial to recap them at this point; one way is to build into the language high-level associative search facilities such as those of LEAP, and the other is to give

the user lower-level mechanisms such as the Search List instruction in DAME, which systematically give the user the next element of the list being searched and test to see if the user wishes to terminate the search or not. If the first facility is provided, then clearly the language must possess a fairly sophisticated list-search mechanism. In such a case, the system might as well give the user the second, lower-level ability too, since this would be at almost no additional cost to the system and there will probably be a number of cases where this lower-level ability will be much more useful or efficient for the user.

(vii) The analysis program text and possibly its internal representation will be of interest to the user in such cases as when he wants to see the texts of the actions associated with a particular type of access to a location or to edit or patch an existing analysis routine. Thus, it is important that the analysis facility contain an on-line editor for analysis text which can also be invoked under program control.

(viii) In addition to accessing the text of analysis routines, the user should be able to access a list of the names of analysis actions associated with a particular address or contact point. This is important, for example, in avoiding duplicate entries for the same analysis routine or in determining in what order the actions associated with an address or contact point should be arranged, e.g. to optimize the set of analysis actions.

Clearly, if the syntax of the analysis language and the form of these associations fall into one of the dominant data types handled by the analysis language, very little additional machinery will be necessary to give the user the abilities mentioned above.

(ix) In the course of analysis computations, the user will often want to hold temporary results in local (or transient) variables. Depending on the kinds of entities manipulated in the computation (e.g. lists, arrays, strings), the user will need to create, and later delete, entities of appropriate type for this purpose. In a highly modularized style of programming, such as we expect analysis programming to be, it is very desirable to have local variables, if for no other reason than the very practical one that whenever one defines a new variable, one would like to be sure that one is not clobbering an already existing variable with the same name, which may have been defined by any one of the number of routines used in the computation. Thus, through the use of local variables, painful searches of all the used analysis routines for each new identifier to be created can be eliminated.

6.3 Continuous Evaluation of Expressions

One of the main functions of a high-level execution analysis language should be to facilitate the description of execution events which the user wishes to watch for. These events can generally be expressed as a change in the value of a predicate from FALSE to TRUE. Such a predicate can involve arbitrary functions over the data elements discussed in the previous section. The continuous monitoring of predicates was discussed by D.Fisher in his thesis [Fi 70]. In this and the next section, I discuss the basic procedure for implementing continuously evaluated expressions as described by Fisher, as well as some points not directly addressed by him.

I shall start by discussing the overall control flow in the continuous evaluation of a set of predicates, deferring the discussion of efficient techniques for the continuous evaluation of individual predicates until the next section.

Normally, when one of these events takes place (i.e. the value of one of the monitored predicates becomes TRUE), some action is taken. Then the question arises: "Should the same predicate be now re-evaluated, because the action may have changed its value again?" More generally, the question is: "What should be the control structure for the continuous evaluation of a set of predicates?" I shall denote by S:⟨predicate⟩ · ⟨action⟩ the specification S that ⟨action⟩ has to be executed when ⟨predicate⟩ becomes TRUE. Consider for example the following specification:

A:(b>0) · (b ← b+1)

where b is an analysis system (not object machine) variable. This specification will cause no changes in the state of the analysis system until b exceeds zero. But after the first time the predicate is found to be TRUE, what happens to the system depends on whether or not the predicate is evaluated again immediately following the action b ← b+1. If it is, clearly the system will fall into an infinite loop (infinite for all practical purposes, unless this is avoided in some special cases by a quirk in the number representation in the system, e.g. adding one to the largest possible positive integer results in zero, or some similar event). This is clearly a very undesirable situation. If the predicate is not re-evaluated, the infinite loop does not result. However, in order to accomodate the case where the user may wish to continue the "predicate evaluation-action" loop until the predicate returns FALSE, a WHILE ⟨predicate⟩ DO (or an equivalent construct) should be available.

We still have not answered our general question regarding the control structure of the predicate evaluation mechanism in full. We have determined that changes to analysis system variables should not cause a re-evaluation of the predicates. How about changes to the object machine state by analysis actions -- should these changes cause a re-evaluation just as if the change were caused by the execution of the object program itself? If so, we can again have the same infinite loop problem. On the other hand, by ruling this out, we are ruling out an important class of functions which the analysis facility should be able to perform: namely, faithful mimicking, by analysis code, of the effects of a piece of object code. For example, a predicate which tests the contents of object machine location X should be able to be activated whether the contents of X is changed by the user program or a model (in the language of the analysis facility) of that program. Thus, this seems to be a desirable ability to have. However, there are some other points also to be considered: How do we handle changes to several variables? How do we handle multiple changes to the same variable in the action associated with a single predicate?

Considering the multiplicity of requirements that a high-level rule for this purpose would have to satisfy, the best policy seems to be to let the user decide what he wants to do, i.e. give him the ability to test if there are any predicates involving a particular OM variable and, if so, to evaluate those predicates and take any associated actions whenever he chooses to do so. It may be desirable to have a high-level operator to do all of this for a given symbol, e.g. an operator, CHECK(X), may serve this purpose, as in:

```
[1]:(A>5) → (B ← A;CHECK(B);A ← C;C ← D;CHECK(C));
```

where we have applied CHECK to B and C but not to A.

One possible model for this control structure appears to be that of Markov Algorithms. In this model, predicate evaluation is halted after the first predicate with a value of TRUE has been found and the associated action has been taken. Following the next change in the object machine state, predicate evaluation starts again from the top, i.e. with the first predicate.

A second possible model is one in which every predicate is evaluated with every change in object machine state and the actions associated with every predicate whose value is TRUE is executed, in static order.

Either model is feasible for this purpose. However, it is clear that the evaluation of even one (arbitrarily complex) predicate after every change in the object machine state can be

unbearably expensive, unless some very efficient techniques are found to perform the evaluation. In fact, unless such techniques are found, the ideas discussed so far in this section just could not be implemented in a useful way. Thus, while these techniques may seem like "implementation details" to some, they are in fact of the essence, since without them these ideas would not be usable.

6.4 Implementation of Continuously Evaluated Expressions

The first, obvious technique which comes to mind in this connection, is to monitor the accesses into each variable appearing in a predicate and whenever such a variable changes value, to re-evaluate the predicates in which it appears, in the proper order. Thus, for example, in the set of (<predicate> → <action>) pairs:

[1]:((X>5)&((Y+Z)<0)) → f_1 ;

[2]:((X<5)&(W<Y)) → f_2 ;

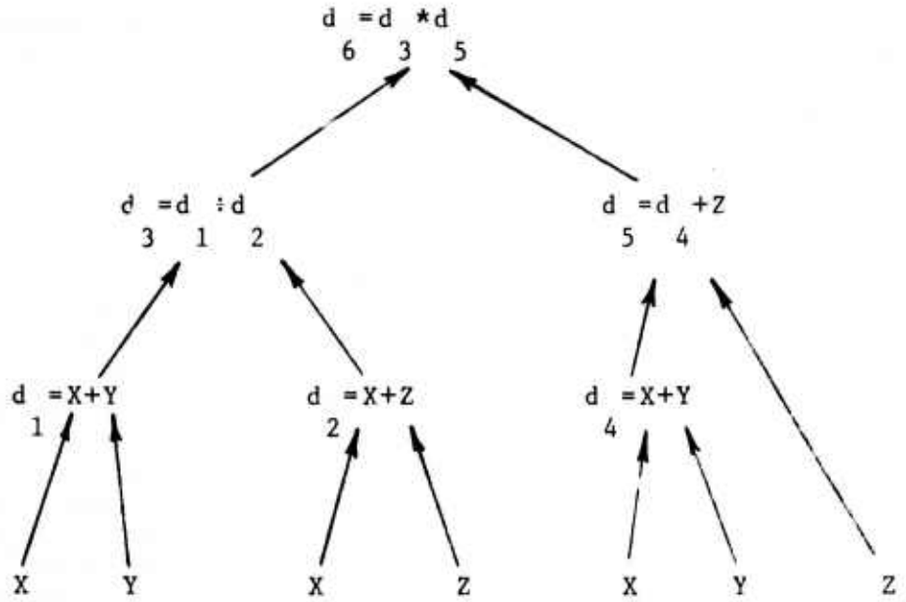
the left-hand sides of [1] and [2] would be evaluated with each modification of X or Y. [1] would also be evaluated with each modification of X, and [2] with each modification of W.

This implementation can be further refined, by observing that whenever the value of a variable V appearing in a predicate P is changed, we do not necessarily need to re-evaluate all of P. First we can evaluate those terms in P in which V appears; and only if one of those terms changes value do we need to make any more evaluations. This suggests a tree representation of the predicate and a generalization of the above evaluation rule to one where a node in level L in the tree is re-evaluated if and only if any of its immediate descendants (in level L+1) is modified.

A further refinement, suggested by D. Fisher [Fi 70], deals with the condition where the same term occurs more than once in a subtree. Consider for example the expression

((X+Y)/(X+Z))*(X+Y+Z)

whose tree representation, following usual operator precedence rules, is given below.



Here, d_i's are dummy variables introduced to hold the current value of each subexpression. Let us suppose that the value of X has just changed and that we wish to propagate that change through the whole expression. If we follow the rule given above in a depth-first, left-to-right fashion, we proceed as follows:

At level 0, we substitute the new value of X into the leftmost instance of X. We evaluate d_1 using the old value of Y. If d_1 changes, we evaluate d_3 using the old value of d_2. If d_3 changes, we evaluate d_6 using the old value of d_5. Then, going back to level 0, we substitute the new value of X into the expression for d_2 and evaluate it using the old value of Z; if the value of d_2 changes, we evaluate d_3 again. If d_3 changes, we evaluate d_6 again using the old value of d_5. Then, after similarly proceeding up the right subtree with the new value of X, we evaluate d_6 a third time.

Thus, in this expression, we have evaluated one subexpression, d_3, twice and another, d_6, three times. In general, if a strict depth-first search is followed, then for each new value assigned to a variable X, every node e in the evaluation tree will be evaluated $n_e(X)$ times, where $n_e(X)$=number of occurrences of X in node e.

Clearly, the reason for the unneccessary computations is the fact that at each node we do not wait for the entire sub-tree under the node to be evaluated. Thus, a breadth-first search for terms to be evaluated, where whenever the value of a term changes as a result of re-evaluation, a flag bit associated with its immediate ancestor is turned on and its own flag bit is turned off, would eliminate the unnecessary evaluations. Thus, in the preceding example, initially all flags would be turned off. Then, when the value of X changed, the flag bits of the X's in level 0 would be turned on. The new value of X will be plugged into each of its occurrences in level 0, also turning on the flags of d_1, d_2 and d_4 (since each is an immediate ancestor of X in level 0). The evaluation would then move to level 1. If the value of either d_1 or d_2 is changed, the flag of d_2 would be turned on. Similarly, if the value of d_4 changed, the flag of d_5 would be turned on. The evaluation would stop either after the root node has been evaluated or when no more flag bits which have been turned on can be found.

This breadth-first search (described by Fisher) thus avoids the unnecessary computations of the earlier depth-first procedure by using an additional bit of information associated with each node of the tree to guide itself to those nodes whose values could possibly change due to the change in one of their ancestors.

Other types of optimizations, such as recognition of common subexpressions (e.g. d_1 and d_4 in above example), could help to further reduce the amount of computation involved.

It must be noted that our ability to determine the leaf nodes of the execution tree which are affected by the change in the value of a variable, X in the above example, depended on our ability to statically locate all the occurrences of that variable in the whole expression. When this is not possible or practical, the above procedure can not be used. Examples of such a case

are function calls or coroutine jumps. In the case of function calls, if the name of the function is statically fixed, e.g. f(X), then one can conceivably locate, at compile-time, the text of the function and see if it uses the variable whose value just changed. If the function call is to a dynamically computed address, this has to be done at run-time, introducing substantial overhead.

Another example of a case where it may be impossible to identify all possible occurrences of a term is accesses to a dynamically computed address. Any term involving the contents of a dynamically computed address should be checked after a change in the value of any object machine variable to see if the dynamically computed address is equal to that of the variable whose value just changed. Alternately, instead of computing the dynamic address with each change in the object machine state, that address itself could be maintained by the continuous evaluation techniques described above.

This last example illustrates a cascaded two-level continuously-evaluated expression and provides an example of hierarchical systems of such expressions as envisioned by Fisher.

CHAPTER 7

EXECUTION ANALYSIS FACILITIES FOR ALGOL-LIKE LANGUAGES

My only hands-on experience with the implementation of the presented ideas on monitoring and modelling has been with programs written at the level of the dominant, contemporary central processor instruction set. In this chapter, I would like to consider the translation of these ideas to the class of languages which has come to be called ALGOL-like languages, which are a subset of "problem-oriented" or "procedural" languages.

It must be emphasized that the intent of this chapter is not to present a design specification for execution analysis facilities for any specific high-level language, but to explore the basic problem areas uniquely associated with this area. Hence the level of detail will be much less than that of Chapter 3 in which the design of a particular prototype system was discussed; but hopefully enough ground will be covered to provide a starting base for the researcher or designer interested in this area.

7.1 The Added Complexity of High-level Languages

In some sense, since a language and its abstract processor, which may be called its "machine", are two sides of the same coin, there should be no conceptual difficulty in translating the techniques we have discussed in the preceding chapters for "machine language" to any language for which a machine exists or can be built. The only difference is that the machines for ALGOL-like languages are much more complex than the machines considered so far. Hence, some things which were very easy to handle before, now become difficult. Let us consider the added complexity in three parts:

1- Syntactic complexity,

2- Semantic complexity,

3- Language implementation complexity.

I shall now proceed much like in Chapters 2 and 3 to discuss these areas first in abstraction, then in reference to a particular language.

7.1.1 On Increased Syntactic Complexity

The increased syntactic complexity arises from the fact that the analysis facility has to be able to understand (parse) each statement in the source program at run-time. For example, one would like to be able to say, at run time: "Trace on the TTY the branch taken by every IF statement", or "type the contents of X and Y[1,2] at entry and exit from any loop in the routine R" or "for every 'else clause' which is executed, type the values of all the variables in the Boolean expression in the associated 'if clause'", or "type the values of the operands of all floating point divide operations, except in routine R".

7.1.2 On Increased Semantic Complexity

An example of added semantic complexity is dealing with scope rules and storage allocation. When specifying a monitoring action on a local variable, one has to qualify the variable name with an identification of the block in which it is declared as a local variable and in which the monitoring action is to be applicable. Further, if the block is executed recursively, then one also has to specify which "generations" or "incarnations" of the variable one is referring to. Similarly, references to the actual parameters of a routine, "own" variables, values returned by expressions etc. must be carefully qualified to ensure reference to the correct data element.

Another example of semantic complexity with some high-level languages is the interpretation of data types; e.g. checking the data types of the actual parameters of a routine inside the routine.

7.1.3 On Complexity due to Language Implementation Techniques

Clearly, an important question which comes up when one tries to envision how such an analysis facility might be implemented is whether the source language is to be interpreted or compiled. A form of compilation called "incremental compilation", in which each statement is compiled as independently of the rest of the program as possible and and control is returned to a run-time monitor after each statement, is a convenient compromise which permits us to reap some of the benefits of both the efficiency of compiled code and the flexibility of interpretation.

To be able to recognize at run-time the code corresponding to the various parts of a source statement in a compiled program (compiled by a non-optimizing compiler), requires some kind of intermediate-level representation of the structure of the object program. This representation may be a directed graph produced at compilation time. Given such a representation of the source

program at run-time, the analysis facility can locate the relevant parts of the object program and arrange for appropriate types of trapping or supervisor call operations when those parts are accessed. Clearly, the symbol table must also be available at run-time. This approach also requires that the structural representation and the symbol table for each external (e.g. library) routine that is used must also be available at run-time.

On the other hand, a "pure interpreter", which essentially re-parses each statement every time it is executed, would not need such an intermediate structural representation, at least in theory, since the effort to parse the program to insert the required hooks is negligible compared to the continuous parsing as part of the interpretive execution. It will of course have the symbol table available which it needs itself during the execution.

Between the extremes of pure compilation and pure interpretation lies a spectrum of modes of execution involving varying degrees of compiled and interpretive representation. In existing languages, these extremens are exemplified, on the pure compilation end, by BLISS, which requires no run-time packages (except I/O, which really isn't a part of the language) and, on the pure interpretation end, by APL, whose right-to-left scanning direction and precedence rule come very close to making it a "post-fix operator" language. In between, lie languages (more accurately, their current implementations) like PL/I, which requires an elaborate run-time library although it is a compiler language, and LISP which can intermix compiled and interpretive execution.

In this multi-dimensional space, for which I can conceive of no significantly helpful metric for the purposes of this research, I shall pick a much smaller subspace in the hope of discussing its dimensions in a somewhat more systematic way. That is the space of purely-interpreted languages. One reason for this choice is that it is conceivable to use an interpreter for any language for program development and testing purposes (or this could be an incremental, debugging compiler). Secondly, this choice permits us to avoid the questions related to code generation and the associated mappings between the source code and the object code. Thus we can concentrate on the functional requirements (in the spirit of Chapter 2) rather than implementation techniques. Thus in the next section, we apply the functional requirements outlined in Chapter 2 and the concepts used in their realization in the DAME system, to the specification of execution analysis facilities for interpreter-based languages.

7.2 Execution Analysis Facilities for Interpreter-based Languages

Let us recall the classes of required capabilities we established in Chapter 2 for a general purpose execution analysis facility:

1- The information to which the analysis facility has access,

2- The points in the execution cycle at which it can gain control,

3- Its instruction set,

4- External appearance and miscellaneous useful features.

The information to which the analysis facility should have access, can be considered in two subclasses:

(i) Information about the execution history of the particular program,

(ii) Information, some may call it "intelligence", about the syntax and the semantics of the source language, as discussed earlier in this section.

Subclass (i) is generically not very different from the corresponding requirement for low-level machine languages; namely, the information needed to efficiently reconstruct any past machine state. In this case, of course, the "machine" is that defined by the source language.

In order to give more concrete content to what is meant by a "machine state" in the case of a high-level machine, let us divide, as we did before, the machine state into two parts: the "state of the memory", i.e. the values of all the variables defined so far, and the "current instruction". The question now becomes: "What is an instruction in a high-level machine?". The question arises because in low-level machine languages, an instruction was an easily identifiable unit, which performed a very small number of, sometimes only one, indivisible operations, usually involving up to three or four operands, including side-effects. Further, what is called a "machine language program" consisted of a sequence of machine instructions. Hence the machine instruction turned out to be a convenient unit for denoting state changes. Clearly, what is usually called a "statement" in a high-level language is not a convenient unit, since it can be arbitrarily long and complex. Thus, it is reasonable to propose the execution of an "operator" as the smallest unit in such a language. However, the "operators"

I have in mind here are a super-set of the operators usually defined in a syntactic description of the language. For example, when in FORTRAN one writes

```
IF(X-Y)10,20,30
```

the selection of the appropriate case as a result of evaluation of (X-Y) must also be regarded as an operator as well as the subtraction. Similarly, the passing of parameters in a subroutine call must also qualify as an operator. Thus, we are led to the intuitive concept of the set of operators in a given source language S as "the set of denotations for the largest operations in S whose effect is indivisible with respect to the semantics of S". This is a very vague and informal definition, in which I rely on the intuitive meanings of all the terms used. The phrase "indivisible with respect to the semantics of S" requires some more elaboration however. Loosely, what is meant by this is that the effect on the machine state of an "operator" can not be broken down into smaller units such that other operators can be inserted between those units. These correspond, in a conventional machine, to those periods in the execution cycle in which the processor is uninterruptible. They also represent, from a user's point of view, the finest degree of detail which can appear in a user request for information.

7.3 A Mini Demonstration Language

For the sake of a more concrete illustration and also to face some of the problems which arise in the application of these ideas, I shall define a very small, hypothetical member of the ALGOL family of languages and use it as a vehicle to explore these ideas further. I have decided to take this approach rather than pick a particular implementation of an existing language since the latter would very likely be a much larger task. In our hypothetical language, we would like the following properties.

(i) It should capture the essence of the syntax and semantics of ALGOL-like languages, i.e. features common to ALGOL 60, PL/I, BLISS, etc. These include ALGOL-like syntax, block structure, recursion.

(ii) Its syntax should be definable by a small grammar (say, about 2 pages),

(iii) Its semantics, similarly, should be easily describable, even if only informally.

The mini-language to be used here was obtained by chopping away a major part of BLISS/10, in fact by removing a great deal of its unique and interesting parts (such as the uniform interpretation of names as addresses, the contents operators, the concept of "structures" etc.), retaining only a small portion which looks sufficiently like ALGOL or PL/I etc. I shall refer to this language as the Mini Demonstration Language (MDL). The syntax of MDL is given in an appendix. For a description of its semantics, I refer the reader to the BLISS/10 manual [WU 71].

7.3.1 Information Accessible by the MDL Analysis Facility

The outline given below follows the one given in Section 6.2:

(i) The external state of the MDL machine, i.e. the contents of every variable and address directly accessible by the MDL program,

(ii) Parts of the internal state of the MDL machine containing the components of the current expression being evaluated,

(iii) The text of the MDL program,

(iv) Information about the structure defined over the user program for purposes of analysis (which may not always coincide with the syntactic structure),

(v) Empirical data associated with each component of the structure, collected during execution,

(vi) Control flow history,

(vii) The text of the analysis program,

(viii) The list of analysis actions associated with each contact point,

(ix) Meta-variables holding intermediate results in analysis computations.

Item (i), access to the MDL variables, does not require any more elaboration. Item (ii) and an important extension is discussed in detail in a succeeding subsection. Items (iii) and (vii), namely access to the texts of MDL programs and analysis programs are discussed together in a succeeding section. Items (iv), (v), (vi) and (viii) are discussed together in the next subsection. Item (ix) is discussed in the section on data types for the analysis language.

7.3.1.1 Representation and Accessing of MDL Execution History

This class of information consists basically of control flow history, data flow history and a mapping between these two histories. I will be recalled that in the DAME system this took the form of a node trace table containing node instance descriptors, each of which in turn contained pointers to the associated node object and input/output sets, as well as certain other dynamic data. Clearly, the key concepts which motivated this implementation are those of nodes, node instances and input/output sets. Given a particular set of nodes, the concepts of node instances, input/output sets and an execution history structured in the above manner are directly transportable to MDL. The rules for defining nodes, however, are not as simple as in the case of low-level machine language, where the only requirement was that each node have a unique entry point and a unique exit point. In MDL, since there is no good analogue of a "machine instruction", we must find a "unit of execution" (UOE), which will in fact be the smallest syntactic construct which can be designated as a node.

The expression-orientation of MDL suggests a natural candidate for designation as a UOE: the simplest expressions in the language. These are: a name or a decimal integer. However, to these we must add an element which can act as a unit of computation: namely, expressions involving a single "operator" which can be any one of the arithmetic or Boolean operators, relations, IF-THEN, IF-THEN-ELSE, SELECT-OF-NSET, EXIT, RETURN, the indexing operator [], the routine call, WHILE-DO and INCR-FROM-TO-BY-DO. Clearly, since only a single operator is to be involved, the two latter loop operators can only appear in degenerate form; they either loop zero times or an infinite number of times or they compute a value already given as an operand in the expression, e.g. INCR I FROM J TO K DO Z or INCR I FROM J TO K DO J etc.. (These loop expressions, when used as UOE's, always return -1 according to the semantics of MDL and BLISS since no EXIT operator can be involved.)

Given the above basic definition for the UOE, the extension to the general definition of a node is very natural: namely, any expression sequence in the language having a unique entry point and a unique exit point. This corresponds in the syntactic specification of MDL to the non-terminal "expression sequence", with the requirement that it not contain any escape-expressions except possibly at the end of the node. This selection is in harmony with the intuitive requirement that node definitions should be compatible with the scopes of routines and blocks; it forces the satisfaction of that requirement automatically.

In adapting the concept of input/output sets to higher-level languages, there are certain issues with respect to the representation of the elements of input/output sets which must be resolved. I shall only sketch some solutions to these issues here since they do not seem to be major problems.

One issue is the representation of local variables in I/O sets. Consider, for example, the following MDL code and the defined nodes N1 and N2:

```
Routine R=begin
       local A,B;
N1     A ← C[1]+2;
       B ← D[A]+5;
N2     C[A] ← f(A,B)
       end;
```

The I/O sets of N1 would look like:

$I_{N1} = [(C(1), V_1), (D[V_2], V_3)]$

$O_{N1} = [(R.A, V_2), (R.B, V_4)]$

The I/O sets of N_2 would look like:

$I_{N2} = [(R.A, V_2), (R.B, V_4)]$

$O_{N2} = [(C[R.A], V_5)]$.

Here, we have denoted by V_i, i=1,...,5, the value of C[1], C[1]+2, D[C[1]+2] and D[C[1]+2]+5 in N1, and f(A,B) in N2, respectively. We have also assumed that the function f does not reference any non-local variables. To denote the use of the local variables A and B, we have used the qualified form R.A and R.B where R is the name of the routine in which they are declared. To qualify local variables which are declared in the inner blocks of a routine, one could employ a block-numbering scheme similar to the one used by some Algol compilers. In such a scheme, a qualifying index is added for each static level and blocks in the same static level are numbered sequentially. For example, in the skeletal code:

```
Routine S= begin
           local A;
             .
             .
             .
               begin
               local A,B;
                 .
                 .
                 .
               end:
             .
             .
             .
               begin
               local A;
                 .
                 .
                 .
                   begin
                   local B;
                     .
                     .
                     .
                   end
             end
         end;
```

the local variables could be represented as: R.A, R.1.A, R.1.B. R.2.A, R.2.1.B. However, as the number of levels increases, this notation quickly becomes cumbersome. To overcome this, at the expense of the loss of some information in the notation, we can abbreviate the indices by using only one index which is the ordinal number of the block-head where the variable is declared, without any reference to static levels. Thus, in the above example, we would have: R.1.A, R.2.A, R.2.B, R.3.A, R.4.B. It is clear that while some information is lost, i.e. static level information, no ambiguity arises from the use of this latter representation.

Similar, but less problematic issues arise with respect to other non-global variables, e.g. parameter formals, own variables. These can also be resolved by the use of a qualifying mechanism such as above.

Since recursive routine calls are permitted in MDL, another qualifying mechanism must be introduced to distinguish among different recursive incarnations of the same routine and their locals.

7.3.1.2 Access to the Internal State and Generic References to Expression Sequences

An important facility that an analysis facility for a high-level language must offer is one which permits the user to say something like:

"If I ever do X, then do <action>"

where X is a partial syntactic specification of an expression sequence. For example, X may be: '<name> ← <name>+1', which would mean that <action> will be performed whenever the MDL machine evaluates an expression whose syntactic form fits the given specification. In such specifications, the permitted non-terminals and their syntactic definitions must be those given in the syntactic definition of the language, or they must be specified formally somewhere (e.g. user manual) accessible by the user. The user may also be given a device for new non-terminals to abbreviate possibly long syntactic forms.

E.g.

```
"Define <sum-terms> → <name> ← <e>+<e>;
 Define <prod.terms> → <name> ← <e>*<e>;
 Define <sum-of-products> → <name> ← <prod.term>+<prod.term>;
 Define <prod.-of-sums> → <name> ← <sum-term>*<sum-term>;

 If I ever do <sum-of-products> or <prod.-of-sums> then do...  "
```

Such a facility should also permit the use of terminal symbols and reserved symbols with special meanings, e.g. to indicate relations between values of non-terminals. For example:

```
"If I ever do 'X ← <name>$1+<name>$2*<name>$1'
    then do <action>"
```

would trigger <action> whenever a value computed by adding the product of the values of two variables to the value of one of them is assigned to X.

An extension to the facility for defining new non-terminals leads us to the notion of "templates", which contain "holes" or formal parameters. For example, one could write:

```
"Define template T(X,Y,Z) → 'if Y then Y else Z';
      .
      .
 If I ever do T('A>B','f(A)','f(B)') then do...;
 If I ever do T('(C+5)<0',<e>,<escapeexpression>)
                then do...
      .
      .
            "
```

These definitions would then cause the system to watch for the expression 'if A>B then f(A) else f(B)' and for expressions of the form 'if (C+5<0) then <e> else <escapeexpression>', and take the specified actions upon their occurrence.

7.3.1.3 Access to MDL and MDLAF Texts

The primary reason for access to the texts of MDL and MDLAF programs is the desirability of on-line editing of these programs. The user should not have to terminate the analysis session to make corrections to either of these programs.

A second reason is to be able to analyze, under program control, and optimize the set of actions associated with a contact point (e.g. to eliminate redundancy, to determine unintended dependencies between actions).

A third reason is to facilitate the specification of the user of expressions which are to be monitored.

Thus there seems to be a need for two different types of editors; one is the more conventional, line or character-oriented editor to be used in preparing and editing of MDL and MDLAF texts; the other is a lexeme-oriented editor which knows the syntax of MDLAF and can respond to requests like:

"If there are any assignment operators in the MDLAF actions associated with fetches from X, return a list of pointers to those actions, else return a null list,"

or,

"Are there any continuously-evaluated MDLAF expressions involving Y? If so, delete them."

It is clear that such an editor will have to know the syntax of MDLAF as well as its internal representation in order to be able to find the desired pointers, delete expressions and the like.

7.3.2 Contact Points and Hook Insertion in the MDL Analysis Facility

Recalling our earlier definition of contact points in the context of low-level machines, as "those points in the instruction cycle at which the analysis facility can gain control", we can translate this notion to the domain of high-level languages such as MDL in terms of the unit of execution which we have selected, namely, individual operands and operators. That is to say, the MDL machine will check for any required monitoring actions after the fetch of each operand of an operator, just prior to and just after the application of the operator, and just before the storage of the result. This requires that we specify the order in which the checks will be made within the expression involving the fetch of several operands. It seems natural that this order should be the same as that which is specified in the language for evaluation of the operands of expressions. BLISS/10, and MDL, give "no guarantee regarding the order in which a simpleexpression is evaluated other than that provided by precedence and nesting..." (BLISS Reference Manual, Jan. 15, 1970, p. 2.2b). Hence, in the expression

```
B ← (C ← 3)+(A ← 5)
```

no guarantee is made about which of the two paranthesized expressions is evaluated and checked for hooks first. However, it is guaranteed that store-hooks for B will be checked after both of the assignments to C and A. I shall denote store-hooks and fetch-hooks for a location X by SHOOK(X) and FHOOK(X) respectively.

In addition to the store-hooks associated with A, B and C, general hooks associated with the initiation and completion of every expression evaluation, which I shall designate by IEXPHOOK and CEXPHOOK respectively, and hooks for the initiation and completion of each specific expression, to be designated by ISEXPHOOK and CSEXPHOOK will be checked. Thus, the sequence of actions in the evaluation of the above expression will be as follows (Note: action sequences seperated by // should be assumed to be done in random order):

```
((IEXPHOOK;
  ISEXPHOOK;
  SHOOK(C);
  C ← 3;
  CSEXPHOOK;
  CEXPHOOK);//
```

```
(IEXPHOOK;
 ISEXPHOOK;
 SHOOK(A);
 A ← 5;
 CSEXPHOOK;
 CEXPHOOK));

IEXPHOOK;
ISEXPHOOK;
d ← C+A;
 1
CSEXPHOOK;
CEXPHOOK;
IEXPHOOK;
ISEXPHOOK;
SHOOK(B);
B ← d ;
     1
CSEXPHOOK;
CEXPHOOK;
```

The above picture probably conveys an exaggerated impression of the overhead involved in checking for hooks. The check for a general hook (i.e. IEXPHOOK, CEXPHOOK) can be as simple as a test on a statically addressable bit and the check for each specific hook (i.e. ISEXPHOOK, SHOOK, CSEXPHOOK) can be as simple as a test on an indirect addressed bit. However, this overhead is still very high if performed by a conventional software interpreter, although perhaps not prohibitive. Therefore, the ideas on microprogrammed and hardware implementations of monitoring and dynamic analysis facilities presented in the next chapter should be studied seriously by interested designers, if such implementations are in the realm of possibilities for them.

7.3.3 An Outline of the MDL Analysis Facility Language (AFL)

AFL is an extension of MDL containing a new data type, several new syntactic constructs and a set of built-in functions and reserved words. In this subsection, these extensions to MDL are outlined.

(i) NODE Declaration

Syntax: NODE <node-declaration list>;

<node-declaration list> → <node-decl.> /
<node-declaration list>,<node-decl.>
<node-decl.> → <node name>=<delimiter>

```
<delimiter> → <routine name> / <label> /
              <routine name>,<block delimiter>/
              <label>,<block delimiter>

<block delimiter> → <integer> /
              <integer>.<block delimiter> /
              <block delimiter>,<block delimiter> /
              <block delimiter>:<block delimiter>
```

Examples:
```
NODE N1=ROUTINE1;
NODE N2=ROUTINE,5;
NODE N3=LOOP,1.1.2.3:5;
NODE N4=LOOP,1.2:3,N5=LOOP,1.4:7;
```

Effect: The indices which are not followed or preceded by a ":", represent lexical levels in the code which is in the scope of the <routine name> or <label>. If a pair <x>:<y> is not present, the entire level is assumed, otherwise the <x>th complete expression through the <y>th complete expression at the level of the last index is defined as a node with name <node name>. Nodes must be disjoint or properly nested. Also, they must begin and end in the same level and block.

(ii) Built-in Functions and Reserved Words

Locating Nodes and Node Instances

Find Node: $FN(<node-decl>)

If no node has been defined which satisfies <node-decl>, returns 0 else returns the address of the first such node object.

Find Node Instance: $FNI(<node-inst. spec>[<node-expr>])

```
<node-inst. spec> → (<MDL expression>)

<node-expr> → <node name>/<node obj. ptr>/
              $NODEOBJ(<node-inst expr>)/
              @<MDL expr>/$CURNODE

<node-inst expr> → $CURINST/$LASTINST(<node-expr>
              [,<node-inst expr>])/
              $FIRSTINST(<node-expr>[,<node-inst expr>])/
              $NEXTINST(<node-inst expr>[,<count>
                    [,<node-expr>]])/
              $PRECINST(<node-inst>[,<count>
                              [,<node-expr>]])/
              @<MDL expr>
```

Effect: Let the value of <node-inst spec> be N. If <node-expr> has been specified, then only the instances of that node, otherwise all node instances are searched. If N>0, then the direction of search is forward in time starting with first node instance; if N<0, the direction of search is backward in time starting with the last instance, with N=0 representing the last instance. If the value of <node-expr> is zero, then zero is returned; otherwise the value is taken as the address of a node-object.

Node Object of: $NODEOBJ(<node-inst expr>)

If <node-inst expr> points to a node instance, then the address of the node object associated with that instance, otherwise zero is returned.

Last Instance of: $LASTINST(<node expr>[,<node-inst expr>])

If second argument is omitted, it is equivalent to $FNI(0,<node-expr>), otherwise a pointer to the last instance of <node-expr> prior to <node-inst expr> is returned. If no such instance can be found, zero is returned.

First Instance of: $FIRSTINST(<node-expr>[,<node-inst expr>])

If second argument is omitted, it is equivalent to $FNI(1,<node-expr>). Otherwise a pointer to the first instance of <node-expr> after <node-inst expr> is returned. If no such instance can be found, zero is returned.

Next Instance of: $NEXTINST(<node-inst expr>[,<count>[,<node-expr>]])

If the 2. and 3. arguments are omitted, it is equivalent to $FIRSTINST($NODEOBJ(<node-inst expr>), <node-inst expr>). If only the 3. argument is omitted, it is equivalent to FNI(<count>+1, $NODEOBJ(<node-inst expr>)). Otherwise a pointer to the nth instance of <node-expr> after <node-inst expr>, where n=<count>, is returned. If no such instance can be found, 0 is returned.

Preceding Instance of: $PPECINST(<node-inst expr>[,<count>[,<node-expr>]])

If the 2. and 3. arguments are omitted, it is equivalent to $LASTINST($NODEOBJ(<node-inst expr>), <node-inst expr>). If only the 3. argument is omitted, it is equivalent to FNI(<count>-1, $NODEOBJ(<node-inst expr>)). Otherwise a pointer to nth previous instance of <node-expr> relative to <node-inst expr> is returned. If no such instance can be found, 0 is returned.

Current Node Instance: $CURINST

A global variable which always points to the node instance which was entered most recently.

Current Node Object: $CUROBJ

Equivalent to $NODEOBJ($CURINST).

Locating Input/Output Sets

Input Set List of Node: $ISL(<node-expr>)

Returns a pointer to Input Set List of node <node-expr> or, in case of errors, zero.

Output Set List of Node: $OSL(<node-expr>)

Analogous to $ISL.

Input Set of Node Instance: $IS(<node-inst expr>)

Returns a pointer to input set of <node-inst expr>, or, in case or errors, zero.

Output Set of Node Instance: $OS(<node-inst expr>),

Analogous to $IS.

Accessing Values of Addresses in I/O Sets

Value-part of I/O set element: $VAL(<id-expr>,<I/O set ptr>, <flag>)

<id-expr> → <name>/<routine name>,<name>/<routine name>, <block id>,<name>

<block id> → <pos. decimal>/<block id>,<pos. decimal>

Indirect Addressing Operator: @<name>

Returns a pointer to the object whose address is equal to the value of <name>.

Get Attribute Value: $GATTR(<obj. expr>,<attr. name>, <flag var>)

Looks for an attribute named <attr. name> in the object <obj. expr>. If such an attribute is not found, <flag var> is

set to zero and zero is returned. Otherwise <flag var> is set to 1 and the value of the attribute is returned.

Add Attribute: $ATTR(<obj. expr>,<attr. name>,<value>)

Change Attribute: $CATTR(<obj. expr>,<attr. name>,<value>)

Delete Attribute: $DATTR(<obj. expr>,<attr. name>)

These functions work in obvious ways. They return 1 if successful, 0 otherwise.

In addition to these functions, a set of conventional list processing functions such as create-list, include-in-list, remove-object-from-list, head-of-list, tail-of-list, cardinality-of-list etc. should be provided.

(iii) Editing MDL and AFL Texts

I shall comment only briefly about this aspect of the analysis facility. The need for two different types of editing abilities has been noted in Chapter 6. One of these is the normal set of functions provided with conventional on-line text editors for preparing and modifying program text. The other, and the more interesting one for our purposes, is a lexeme-oriented, rather than character or line-oriented, editor which can work on list-structure representations of MDL and AFL parse trees. Considerable work has been done on such a syntax-driven editor by L. Robinson and D. Parnas ([RP 73] and [Ro 73]). I feel I can do nothing better than cite these references here.

(iv) Explicit Hook Insertion

(iv a) Monitoring of Accesses to Variables

To monitor the accesses to a variable explicitly, AFL contains the ON FETCH, ON STORE and ON USE facilities, whose syntax is:

<hook name>: ON <condition><varlist> DO <expr>;

or,

<hook name>: ON <condition> DO <expr>;
<condition> → FETCH/STORE/USE
<varlist> → <MDL variable id>/
<MDL array id>[<index expr>]/
<varlist>,<varlist>

For example, "H: ON FETCH X DO expr " will cause the evaluation of expr whenever the contents of X are fetched in the evaluation of some MDL expression. If X is omitted, expr will be evaluated with the fetch of every operand of every expression. $OPDADDR and $OPDVAL will contain the address and the value of the current operand.

ON STORE and ON STORE X work similarly, except that they are checked prior to store operations.

ON USE and ON USE X cause checking upon both fetch and store operations.

(iv b) Monitoring of Expressions

Expressions to be monitored can be specified in one of two ways: by lexical location or by giving the syntax of the expression. Further, the monitoring actions can be specified to be taken just before the application of the "root" operator of the MDL expression or just after it.

Specification by Lexical Location:

<hook name>: BEFORE <MDL location list> DO <expr>

<hook name>: AFTER <MDL location list> DO <expr>

<MDL location list> → <MDL location>/
<MDL location list>,<MDL location>

<MDL location> → <delimiter>

(See the syntax of the non-terminal <delimiter> in the second paragraph of 7.3.3)

Examples:

```
L:  BEFORE ROUTINE1, ROUTINE3 DO(X ← Y+1;TYPE(X));

L1: AFTER LABEL1.1.3 DO $DISAB(L);

L2: AFTER LABEL1.1.3:5 DO $TYPE(Z);
```

The first example will cause the paranthesized sequence of expressions to be evaluated before every call on the routines ROUTINE1 and ROUTINE3, after their parameters, if any, have been evaluated.

The second example will disable the above action after the third expression in the first block (or expression) following and in the same block (or compound expression) and level as LABEL1.

The third example will cause the value of Z to be typed out after the evaluation of each of the 3., 4. and 5. expressions at the top level of the block (or compound expression) mentioned above.

Specification by Syntax:

<hook name>: BEFORE EACH <syntax spec.> DO <expr>;

<hook name>: AFTER EACH <syntax spec.> DO <expr>;

<syntax spec.> takes the form of an expression in which non-terminal symbols enclosed within <,> or the special symbol $* (it means "anything") may appear.

Examples:

L: BEFORE EACH <loopexpression> DO <expr>;

L1: AFTER EACH $*+A DO <expr>;

The first example would cause the evaluation of <expr> before the evaluation of any WHILE and INCR expressions. The second example would cause the evaluation of <expr> after each addition operation involving A as the right-hand operand.

The syntactic specification facility could be extended by implementing the features (involving "templates" and user-defined non-terminals) discussed in subsection 7.3.1.2. I shall not dwell on these extensions here.

(iv c) Monitoring of the Control Path

There are two facilities for monitoring the flow of control, in addition to the BEFORE and AFTER features described earlier. These are:

<hook name>: ALONG PATH <path descriptor> DO <expr>;

and

<hook name>: AFTER PATH <path descriptor> DO <expr>;

<path descriptor> → <delimiter>/
<path descriptor>,<delimiter>

<delimiter> → <unit>/<unit>(<count>)/<unit>
[<path descriptor>]

<unit> → <routine name>/<label>/<node name>

The first expression causes <path descriptor>, which contains, say, n <delimiters>, to be matched continuously against the control path. If, for some k≤n, the first k elements of <path descriptor> match the most recent k elements of the control path (which are routine or node names or labels), then <expr> is evaluated.

The second expression causes <expr> to be evaluated only at the completion of the specified path.

In the specification of <delimiter>, the option <unit>(<count>) means that <count> number of consecutive executions of the same <unit>, without the intervention of any other unit s, is to be watched for and treated as a single element in the path. The option [<path descriptor>] provides for nesting of paths.

Examples:

L: ALONG PATH ROUT1[ROUT2, LABEL1, LOOP1], ROUT3 DO <expr>;

L: AFTER PATH R1(2),R2[R3(4),R4[R5,R6]](3) DO <expr>;

In the first example, <expr> will be evaluated after the execution of each of ROUT2, LABEL1 and LOOP1 inside ROUT1, after exit from such an execution of ROUT1 and after ROUT3, provided they occur in that order with no intervening <unit>s.

In the second example, the interpretation is similar, except that multiple consecutive executions of certain <unit>s are to be considered as single elements.

(v) Continuously Evaluated Expressions

CSELECT <elist> OF CSET <cexpressionset> TESC

<cexpressionset> → /<ce>/
<cexpressionset>;<ce>

<ce> → <MDL expression>: <AFL expression>

<elist> → <AFL expression>/<elist>,<AFL expression>

As will be obvious to those familiar with BLISS, this syntax follows the syntax of the SELECT expression in BLISS, and hence the expressions defined by it are called CSELECT (for "Continuous Select") expressions. Its evaluation can be precisely described by saying that it is equivalent to the evaluation of the AFL expression "SELECT <elist> OF NSET <cexpressionset> TESN" after each change in the value of any MDL variable in <elist> or in the left-hand sides of <cexpressionset>.

Example:

```
CSELECT (D+E) OF CSET

        A-B:f1 ;

        A*B:f2 ;

        C+D:f3

        TESC;
```

This example will cause the monitoring of the values of A,B,C,D and E and the continuous updating of the values of the expressions D+E, A-B, A*B and C+D. When the value of D+E changes, the value of the first left-hand expression, A-B, is compared with the new value of D+E. If the two values are found equal, then the expression f_1 is evaluated. Then, the next left-hand expression, A*B, is compared with D+E, and if equal, f_2 is evaluated. This process continues, until all left-hand expressions have been tested.

Important note: if the value of a left-hand side is equal to the value of the controlling expression (D+E in this example), the right-hand side will be evaluated with each change in the value of an MDL variable, until the values of the left-hand side and the controlling expression become unequal.

CHAPTER 8

ARCHITECTURAL FEATURES FOR EXECUTION ANALYSIS

As has been previously noted, one of the major impediments to the wide use of the kinds of simulator-based techniques described so far is the slowness of simulation at the memory cycle level and the information loss incurred with simulation at instruction level. Further, if, unlike in the DAME system, the object machine and the host machine are the same, then one would like to be able to execute the uninteresting parts of the program, i.e. the parts we do not wish to include in the analysis, at full machine speed and only incur overhead over the monitored parts. This becomes an important factor, for example, in the case of trying to isolate a bug which appears only after a considerable amount of execution.

To deal with this problem, we have to design architectural features to be implemented in hardware or microprogram which would significantly reduce the amount of monitoring done by software. Thus, in this chapter, I shall discuss:

(i) Various techniques for the implementation of the hook mechanism as a function of the relative word lengths of the host machine (W_H) and the object machine (W_O),

(ii) The implementation of the node mechanism, in particular the node objects, the node trace table and the input/output sets, along with the types of storage technologies appropriate for these data structures,

(iii) The interface between the host machine and the object machine, in particular the data paths and the control paths between the two.

I shall then conclude the chapter with an outline of a unified architecture embodying the various features discussed, assuming a simple, conventional CPU architecture for the object machine, and a review of several reports on hardware and microprogrammed measuring and monitoring facilities by other workers.

8.1 The Hook Mechanism

The three operations which lie at the heart of any monitoring scheme are: (i) Given a particular set of contact points in the course of the execution of the object machine, the determination of whether there is any monitoring action to be taken at the current contact point, (ii) If so, locating the description of the action to be taken, (iii) Taking the desired action.

Step (i) clearly has to be done continuously, i.e. at every occurrence of a contact point. This is the basic price paid for running on a monitored machine. Therefore, it is desirable to minimize this overhead. Step (ii) is normally performed much less frequently than step (i). Thus, in programs which are not heavily monitored this step will not normally cause excessive amounts of overhead. In heavily monitored programs however, this step can cause sufficient degradation of performance to prevent wide spread use of the monitoring facility. The amount of overhead caused by step(iii), of course, is a direct function of the particular actions to be taken and of their execution by the analysis facility. In the rest of this chapter, I shall explore several techniques for implementing these operations in conventional single-instruction-stream/single-data-stream processors. For this purpose, let us distinguish three cases:

(i) The host machine has a longer word-length than the object machine ($W_H > W_O$),

(ii) The word lengths of the two machine are equal ($W_H = W_O$),

(iii) The host machine has a shorter word-length ($W_H < W_O$).

8.1.1 Monitoring with W_H Greater than W_O

As already discussed in Section 3.2, the availability of extra bits in the host machine word greatly facilitates the monitoring operations mentioned above.

Machine architectures with this feature are also known as "Tagged Architectures". Many applications of this architecture, including some which are not discussed in this thesis, were discussed by E. A. Feustel in his paper "On the Advantages of Tagged Architecture" ([Fe 73]).

Depending on the number, n, of extra bits available, one can use them as flags (say, with $1 \leq n \leq 8$) or as indices into a table ($9 \leq n \leq A_H$), or as an address in the address space of the host machine ($n \leq A_H$), where A_H is the width of a host machine address. Let us, then, first consider the use of flags for this purpose.

"Flag-bit" Implementation

One approach may be as follows: one designates a flag bit for each kind of contact point applicable to addresses (as opposed to general contact points). These contact points may be, for example, designated as in the DAME system: namely, (i) after every fetch, (ii) before every store, (iii) after every instruction fetch, (iv) after every instruction completion. In this case, one would need four bits. If fewer than four bits are available, then one can combine some of the flags and implement a flag in the CPU indicating the type of operation currently being performed. In such a case, the monitoring logic would test the conjunction of the flag bit in the current word being fetched or stored, and the CPU operation flag. For example, with three bits instead of four, one can combine the fetch and the instruction-fetch flag bits. There would be a bit (let us call it the I-bit), indicating whether or not the current fetch cycle is an instruction fetch cycle or a data fetch cycle. This bit has to be accessible by the monitor routines. Then, the user who wishes to detect the accesses to a particular location as an instruction-fetch, would insert a hook to be activated upon every fetch from that location, and within that hook, test the I-bit to determine if the current access is an instruction fetch or not. Since, usually, the same word is not accessed both as data and as instruction, this technique would involve a conjunction and a comparison as an overhead only in fetches from locations containing an instruction. This does not seem to be an excessive price to pay. In fact, if one is sure that the location being hooked is always accessed properly, one can eliminate this test altogether. This will probably be the most common case.

A problem arises in certain computers however, if one wishes to insert a hook in every instruction word of a large block of consecutive locations. A case in point is the PDP-11, which contains in-line data interspersed with instructions involving certain addressing modes. Since, in general, it is impossible to tell statically if a particular word contains data or instruction, the insertion of hooks only in locations containing instructions can not be mechanized, i.e. the user has to either hook each instruction word individually, or hook all the locations in a given block (using a mechanism similar to the DAME HOOK command

which accepts an address range as a parameter) and then go in and delete the hooks for individual locations containing in-line data. Either way, it is a fairly painful process. An easier method would be to perform a test in the monitor routine to see if the current cycle is an instruction fetch cycle or not.

Thus, assuming that through the use of some combination of flag bits, the presence of some monitor action to be taken at a contact point can be determined, let us now consider the problem of locating the description of the monitor action to be taken. Ignoring the format and syntax of monitor actions themselves, I shall assume that a single pointer is sufficient to locate the desired action description. Hence, what is needed is a table look-up procedure with two inputs, the current address and cycle (i.e. instruction or data), and one output, a pointer to the desired monitor action description. I shall not elaborate on the implementation of this procedure; two most obvious approaches which come to mind are via an associative memory or via a micro-programmed table-lookup mechanism.

"Table Index" Implementation

Let us now consider the case where W_H is sufficiently larger than W_O to permit the insertion of an index for a table, M, into each host machine word representing an object machine word, in addition to, or instead of, the flag bits. In this case, each entry in table M would contain either the description of the action itself, or a pointer to it. Thus, we would not need an associative memory or microprogrammed look-up procedure, since the table index would be built into each host machine word.

The limitation of this approach of course is that if there are k bits available to be used as an index, one could have at most a 2^k element direct-access table. Such a table could be extended by chaining overflow areas to each entry etc. at the cost of some more search.

"Full Pointer" Implementation

If the number of bits available is greater than or equal to the address width of the host machine, then one can in fact store there the full address of the monitor action description. This eliminates the need for a pre-allocated table to contain the action descriptions or the pointers to them. It permits a list-oriented structure to be created and maintained dynamically. (As will be recalled from Chapter 3, DAMF goes one step further

and creates a general list of "interesting objects" for each location requiring one, e.g. such locations as node entry points, or addresses whose previous values are being collected. Pointers to monitor actions, i.e. hook objects, are simply inserted and deleted as elements in these lists as required.)

8.1.2 Monitoring with W_H Equal to W_O

This includes the important special case where the object machine and the host machine are the same. Hence, it will be discussed in some detail.

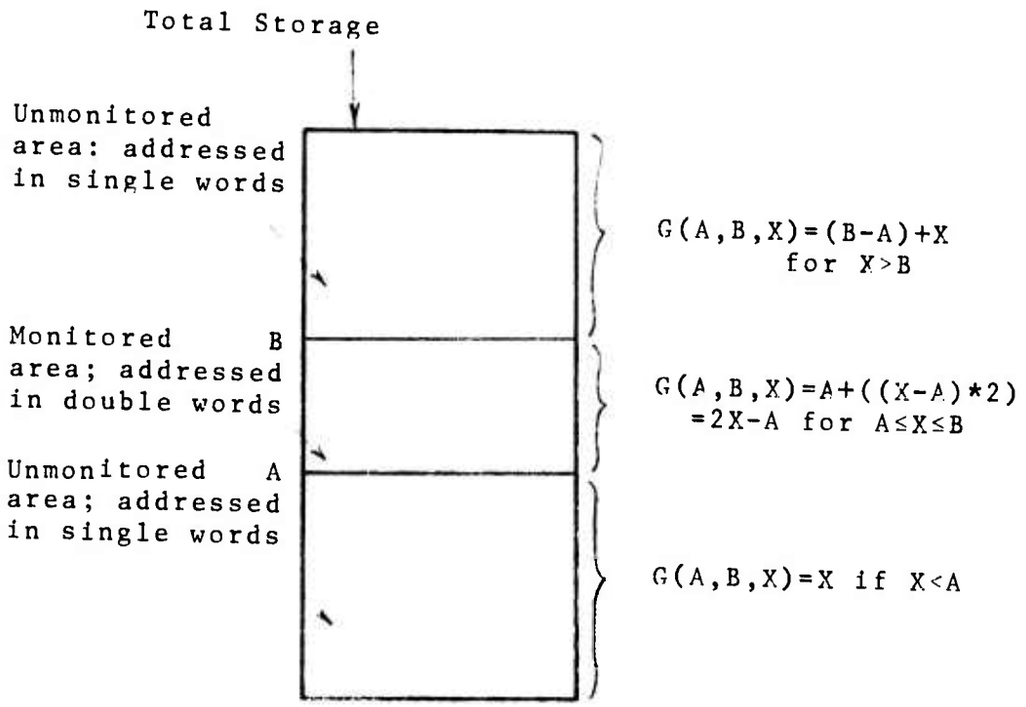
Here we have, for each memory access, two pieces of information with which to determine whether or not the address being accessed is being monitored and, if so, to locate the monitor action description: namely, the object machine address and its contents. A technique of obtaining this information by using only the address has already been discussed above. Another technique which uses both the contents of the accessed address and the address itself, called "Lambda monitoring" [LA 72], was described in Section 3.7.1. I shall summarize this technique here again. The Lambda monitoring technique relies on finding a bit pattern, Lambda, which is expected to be used very rarely by any object program as instruction, address or data. Lambda can be determined by the user at load-time (if he wishes to use a different pattern than the default one) and kept by the system in a Pattern Register. Each data element fetched from the main memory or a register would be compared with Lambda and a monitor trap would be caused whenever an object machine location containing that pattern is accessed. Clearly, this operation should be quite transparent to the user program and the actual contents of that address should be made available to the user program by the control logic upon completion of the monitor action. Once a monitor trap is detected, one then has to locate the associated monitor action description. For this purpose, again, the object machine address being accessed could be used as an input into an associative memory or microprogrammed look-up procedure to obtain a pointer to the monitor action description. If the bit pattern Lambda is the actual contents of the accessed address, then the table search mechanism would return a "no-hit" code which would terminate the trap. One can generalize this technique somewhat by defining several bit patterns, to be kept in different pattern registers, indicating different kinds of monitor traps, e.g. one for each hook type, provided one can find several patterns which are likely to be used very infrequently by user programs. This would enable one to search a unique, and hence smaller, table for each such bit pattern.

The main potential difference in the performance of the first technique, i.e. looking up every generated address in a table, and the Lambda Monitoring technique depends on two criteria: (i) how well the table look-up procedure can be overlappe with the normal object machine operand fetch procedure, and (ii) how often the bit pattern Lambda is used by the user program We must recall here that the terms like "operand fetch" and "operand store" are to be interpreted very liberally and that they refer to every register and main memory location addressable by the user program. In register-oriented machines (as most current central processors are), most references are to registers and not main memory. Hence, the overlapping of the table look-up procedure, mentioned in criterion (i) above, is with respect to the shorter one of the CPU cycle and the main memory read time. If there is a substantial un-overlapped portion, then the first technique is apt to be much slower than the Lambda Monitoring scheme. On the other hand, if the bit pattern Lambda is poorly chosen and crops up often in the user program as data, address or instruction, then the overhead associated with the latter scheme can approach that of the former.

In addition to these two techniques, a third one can be envisioned. This technique essentially is an architectural feature which permits a modification of the physical addressing structure of the object machine to make the host machine word length H longer than the object machine word length W_O, in a way that is transparent to user programs, i.e. retaining the logical addressing structure, except that the amount on physical main memory available would be reduced. For example, consider a machine architecture in which there are two modes of operation: (i) normal user production-run mode, (ii) analysis mode. In the normal mode, the machine functions with no changes to the addressing structure and instruction execution. In the analysis mode however, every user-generated address is multiplied by two, and the contents of the double-word at that address is taken. The lower half of that double-word represents the word which the user is trying to access, and the upper half holds a pointer to the monitor action description, much like in the "full pointer" implementation discussed in the preceding sub-section. These two halves can be retrieved either sequentially, using the same memory port, or in parallel using a seperate memory port for the two words. In either case, the monitoring facility would pick up the monitor word and perform the described action, if any. This technique trades off half of the storage of the object machine for the avoidance of a table-lookup procedure, by in effect using the current object machine (OM) address to locate the OM word and the monitor action address simultaneously. Hence it may be an attractive alternative in cases where a great majority of the programs to be analyzed require less than half the available storage for program code and data.

A refinement of this technique, requiring a little more intelligence implemented in the hardware or microcode controlling address generation inside the CPU, permits user control over, and possibly drastic reductions in, the amount of extra storage required. Let us suppose, for example, that the area of main memory which we are interested in monitoring lies between the lower bound A and upper bound B. Then, clearly, a scheme which would multiply by two only those addresses between A and B and still remain transparent to the program being monitored would provide the above advantages. Such a scheme can be implemented in a straightforward manner. Let us define two registers A and B in the machine, whose contents are to be specified by the user at program and data load-time. Then, each user address N generated during loading and execution is tested by hardware to see if it falls between A and B. If so, it is mapped into A+((N-A)*2)=2N-A. If N is less than A, it is mapped into itself. Otherwise (it is larger than B), it is mapped into C+N where C is a constant which is computed when the registers A and B are loaded and is equal to B-A.

This address transformation, denoted by G(A,B,X), is illustrated in the next figure.



This technique can be generalized to the case involving M monitored areas, M>1. Such a generalization requires the comparison, possibly in parallel, of X with the limit registers for each of the monitored areas and the selection of a different constant to be added to X or 2X for each position of the memory. Thus, if there are M such areas, with limits, A_i , B_i , i=1,...,M, and X is found to be in the Kth monitored area, then X is mapped into $A_K + \sum_{j=1}^{K-1} (B_j - A_j) + 2(X - A_K) = (\sum_{j=1}^{K-1} (B_j - A_j) - A_K) + 2X$. If X is smaller than A_1, it is unchanged. If it is in an unmonitored area following the Kth monitored area, then it is mapped into $\sum_{j=1}^{K} (B_j - A_j) + X$.

8.1.3. Monitoring with W_H Less than W_O

This case is conceptually not very different from the preceding two cases; however, its implementation will probably be much more inefficient, especially if W_O is not some integral multiple of W_H. I shall not say much on this case except to point out that it can be made equivalent to either of the two preceding cases by an address-transformation mechanism of the type described earlier. By choosing this transformation suitably, one can make available a number of bits in the representation of each word of the object machine, so that these bits can be used as described under the "Flag bits", "Table index" or "Full pointer" approaches. The basic inefficiency lies in the fact that one has to access several HM words to simulate access to each OM word.

I conclude here the discussion of various techniques for implementing the hook mechanism. The choice of the appropriate technique for a particular processor will depend on the word sizes of the two machines, and the types of memory and microprogramming capability available.

8.2 Implementation of the Node Mechanism

One of the major components of the Analysis Facility is the Node Mechanism which includes: the node objects, the Node Trace table, the input/output sets, and the creation, maintenance and searching of these data structures. Hence, in this section I would like to discuss feasible approaches to the implementation of this mechanism. I shall assume that the Analysis Facility has available for its use a certain amount of main memory, a much smaller amount of high-speed local memory and some associative memory. Data structures will be assigned to the type of memory most appropriate to their size, frequency and method of access. I shall also assume the existence of data paths between the local memory and the main memory, and between the associative memory and the main memory as well as microcode instructions to make transfers along these paths.

The node objects are created when a node is defined. In the course of the execution, they are accessed whenever the corresponding node is entered or exited or when a monitor instruction refers to them. They do not take up very much room, about 8-10 words per object. Except for the current node object, they are normally not accessed very often. Hence, an appropriate storage allocation for node objects would be to keep them in main memory, except for the current node object which will be brought into the local memory when the corresponding node is entered, maintained in the local memory during the current node instance and put back in its earlier position in main memory when the current instance is exited.

The Node Trace table is a dynamically growing structure whose size is a direct function of the number of node instances executed. Here too, normally only the table entry for the current node instance is accessed often. Hence this latter part can be kept in local memory in the same manner as the current node object and the rest of the table can be kept in main memory.

The same remarks also apply to the input/output sets with one important qualification: the maintenance of the current input/output sets will probably be best implemented through an associative memory. This is due to the fact that one has to search the current input or output set for every generated address during the current node instance. If the address is not found, it must be added to current input or output set. If this is done via a sequential search of these sets, the resulting overhead is likely to be unacceptable. Thus, the current input/output sets should be created and built up in associative memory and transferred to main memory and linked to the I/O set list of the associated node when the current node instance is exited.

Another point worth mentioning with respect to the I/O sets is the nesting of these sets if nested nodes are permitted. Suppose there are n levels of nested I/O sets: what is the best way to maintain them - to maintain all of them with each generated address, or to maintain the highest level set only and to update the next highest level (i.e. its parent) only when the former is exited by adding the appropriate entries from the highest level set into the next highest one? Both approaches are feasible. The sophistication of the associative memory available and the overhead of the two approaches will determine the preferable alternative for a particular implementation.

8.3. The Interface between the Analysis Facility and the Central Processor

Since the Analysis Facility requires access to much information inside the CPU and to the main memory, and since it needs the ability to interrupt the CPU, it is worthwhile to consider the interface between the Analysis Facility and a "conventional" central processor. I shall not go to great detail in doing this however; hence I shall not refer to a specific processor, but rather to one which is representative of contemporary architecture.

The interface between the analysis facility and the central processor consists of data and control paths between the analysis facility processor and:

1- Main memory address register (data path), MARP,

2- Main memory data register (data path), MDRP,

3- Main memory access control (control path), MACP,

4- General registers (data path), GRP,

5- Internal registers (data path), IRP,

6- CPU control logic, (control path), CLP.

The first three paths, MARP, MDRP and MACP, permit the analysis facility to access the main memory address and data registers as well as main memory locations. The path GRP gives access to the contents of the general registers.

The path IRP gives access to internal registers which contain information about the opcode and operand fields obtained by the CPU by decoding the current instruction (e.g. for the PDP-11, these would be current opcode, source and destination mode and registers etc.). In the existing design of some processors, this

information may not be explicitly kept in this form during the entire execution cycle. In such a case, either the processor design may be modified to make this information available or an instruction decoder may be built into the Analysis Facility which can extract the required information.

The path, CLP, to the CPU control logic is a control path which serves to synchronize the activities of the Analysis Facility and the CPU. In particular, it will conduct signals from the former to the latter to inhibit and enable instruction execution.

8.4 The Analysis Facility Processor (AFP)

So far, we have said nothing about how the Analysis Facility will function, its instruction set and internal organization. While it is not desirable to go into much detail here, it is probably worthwhile to outline the answers to these questions.

The question of thow the Analysis Facility Processor (AFP) will function, i.e. "will it have its own instruction execution hardware or will it share that of the object machine?", and the question of the instruction set of the AFP are interrelated. Recalling the two subsets of the instruction set of DAME, namely the "conventional" subset and the "monitoring and analysis" subset, it is clear that if the AFP uses the same instruction set as the object machine, then the monitoring and analysis instructions must be compiled into the conventional subset, which can then be directly executed by the AFP.

This approach has the advantage of not requiring a seperate instruction set processor for the AFP. However it also requires that the internal state of the object machine CPU be saved before the AFP can do anything. Also, if the CPU is to be monitorable while it is being used by the AFP, then, in fact, the internal state of the CPU has to be saved in a stack, to permit an orderly return from the various levels of monitoring and analysis activity. Further, the object machine instruction set would have to be extended to permit access to the internal registers, MARP, MDRP etc., perhaps requiring new instructions. Finally, the instruction set of the object machine will not necessarily be suitable to perform the monitoring and analysis actions described in Chapters 2 and 3. In particular, if the list-structure orientation of the DAME system is also adopted in the design of the AFP, one would really prefer to have a machine instruction set suitable for list processing. For these reasons, my preference would be a seperate instruction set processor for the AFP, both from the performance point of view and the freedom it affords in defining new types of operations. Hence, in the design for the AFP whose outline is given in the following illustration, I assume a seperate AFP,

possibly implemented in microcode, using the three types of storage, namely main storage, high-speed local storage and some associative memory as discussed earlier.

It is worthwhile at this point to review several reported implementations or designs with objectives roughly similar to, but all less ambitious than, ours.

Bussell and Koster [BK 70] reported on an experiment involving the instrumentation of programs for the XDS Sigma 7, mainly for the purpose of timing and instruction mix measurements. Probably the most significant part of this paper is a description of a "vernier-scale" technique for measuring instruction and event times to a much higher resolution than the time between the "ticks" of the hardware clock, with a precision limited only by the precision with which the hardware clock ticks. This part of the paper makes it a "must" reading for those interested in program instrumentation. Apart from this technique, the paper contains a good discussion of the overhead of execution under simulation ("simulation artifact") and by using the Execute instruction in the Sigma 7 and in S360/75.

In an earlier paper entitled "SNUPER COMPUTER- a computer in instrumentation automaton", G. Estrin et al describe a design (which, to the best of this author's knowledge, was never implemented) for an automaton connected to the object computer by sensors, which would select user-specified events occurring in the course of the execution and transform this data into a count kept in an element asseciated with that event. The paper, while ahead of its time (1967) in some of the ideas proposed in it, leaves unspecified the crucial questions of (i) how the raw data coming in from the object computer is converted into an index for the bit table whose elements indicate the significance or insignificance of the events bearing that index, (ii) what parts of the state of the object computer (IBM 7094) are accessible by the SNUPER COMPUTER, (iii) what kind of a language will be used by the user to specify the event, graphic displays and the like.

M. Zelkowitz in [Ze 71] describes briefly an associative memory-based design for associating interrupt routines with fetch/store addresses. His design includes a "condition code" field in the associative table entry which indicates which of the "less than", "equal to" and "greater than" relations must hold between the address currently being accessed and the address stored in the table entry in order for the transfer of control to a second address specified in the table entry. Apart from this feature, this design is the basic, straightforward approach to monitoring addresses. However, in this author's opinion, the associative table will need to hold substantially more than the 16 entries

envisioned by Zelkowitz's design. Also, this design does not provide for monitoring accesses to the general registers.

Two other reports on hardware-monitor based approaches should be mentioned here. One is the report on the Neurotron monitor by R. A. Ashenbrenner et al [ALN 71], and the other is the report on a hardware monitor for a multi-mini processor (C.mmp) system by S. H. Fuller et al [FSW 73]. Both of these monitors appear to be oriented toward data selection and collection and not the full spectrum of general purpose, dynamic analysis activities envisioned in this thesis.

The paper which comes closest in spirit and approach to those described in this thesis is that of H. J. Saal and L. J. Shustek [SS 72]. In this paper entitled "Microprogrammed Implementation of Computer Measurement Techniques", the authors report on a project in which a Standard Computer Corporation IC7000 computer, which contains a writable control store, was microprogrammed to collect (i) execution history data by recording all successful branch instructions and relocation information, and (ii) distributions of the usage of operation codes and consecutively executed operation code pairs. What is of interest to us here, is not the actual data or the types of data collected, but rather the insights they provide on the problems of inserting measurement routines in emulators. This paper is also "must" reading for those interested in building microprogrammed instrumentation facilities. Since these problems are so relevant to this discussion, I outline them here:

(i) "... since microprogram storage is an extremely scarce commodity, it was prohibitively expensive to insert measurement routines throughout the microprogram." Thus, in the Analysis Facility Processor, the power of the instruction set might be limited by the microprogram storage available.

(ii) "Since our microcomputers possess a limited subroutining facility at the microprogram level, it was not even feasible to include a subroutine call at every point at which we wished to measure the performance of the system." This is an example of the problems caused by the primitiveness of the microprocessor instruction set. More on the same point: "A severe problem found in the implementation of extensions via microprogramming, generally not found in conventional software interpreters, arises from the lack of many general facilities at the microprogram level."

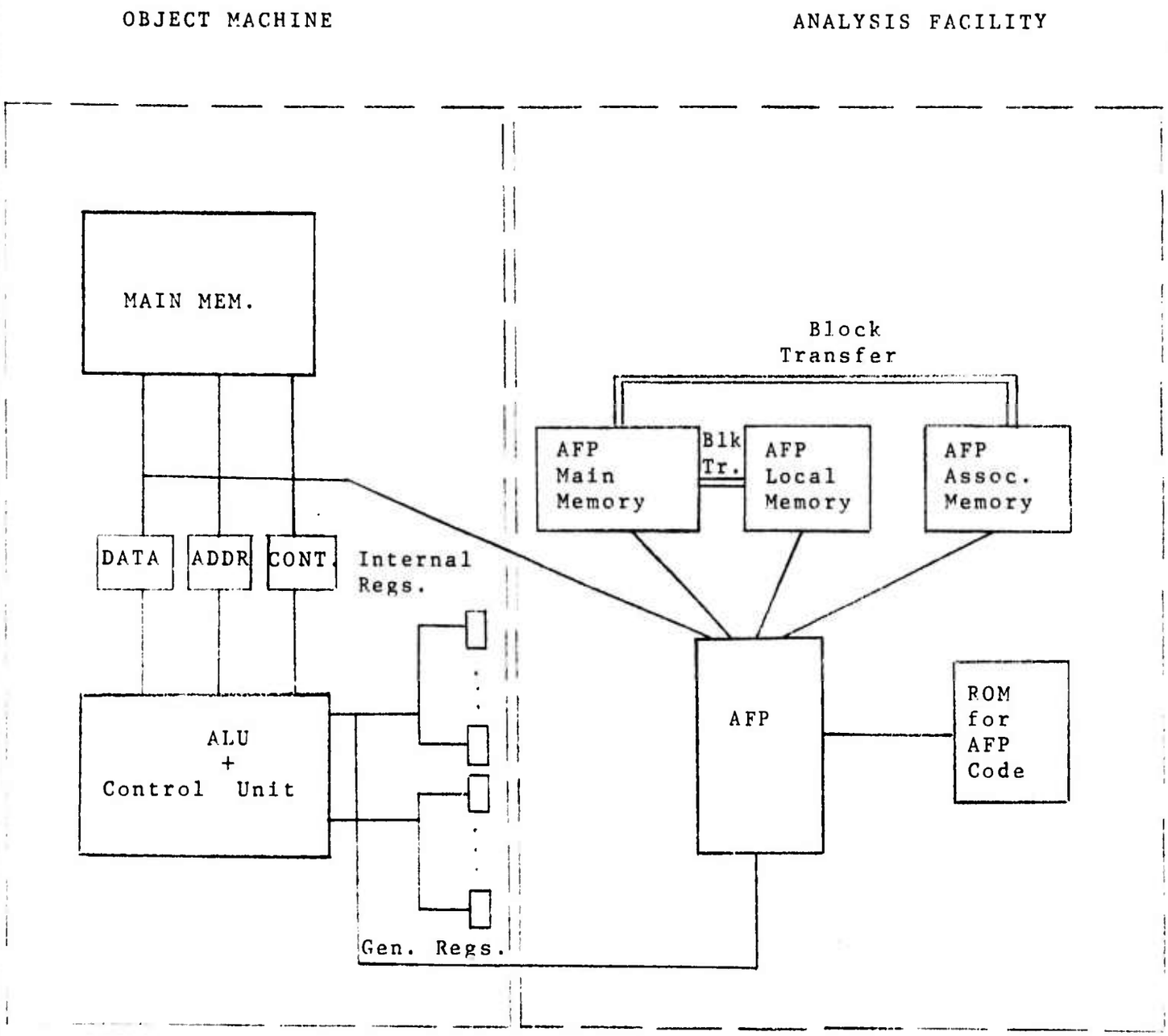
(iii) "In addition, many instructions are executed directly in hardware at instruction fetch time (most of the program transfer instructions). Others share microcode but are semantically distinguished by a large number of flip-flops (set by the hardwired instruction fetch and decode) which perform extensive residual

control." Those flip-flops may well include data about addressing modes, the success of a conditional branch etc. and should be accessible by the measurement routines. More on the problems caused by hardware interpretation: "Microprogram machines are generally not completely microprogrammed. Many aspects of instruction decoding and operand fetching may be performed in a hardwired scheduler in the interest of increased efficiency. This technique conflicts with microprogram measurement. The hardwired decoding scheme may automatically set a variety of residual control registers and flip-flops to simplify the semantic emulation routines. Current microprocessors have not been designed to allow these registers to be explicitly read by an emulator and thus they are not evailable to measurement routines. This lack of generality imposes unnecessary complications to the microprogrammer, but could be avoided in future microprocessor design."

(iv) "The Input/Output conflict between the microprogram measuring routine and the system being measured was the single most difficult problem in the implementation". The authors recommend that the two systems use different channels for input/output.

All of these points are candid examples of the problems which arise in the design and implementation of microprogrammed execution analysis facilities. They emphasize several points already made in the discussion of the AFP above: namely, the need for a powerful instruction set, access to object machine internal registers and seperation of the object machine hardware from that of the AFP.

Illustration 8.1



CONCLUSIONS

The objectives of the research were (i) to explore the possibility of designing facilities which are significantly more helpful to the user in many types of execution analysis than existing systems, (ii) to identify key problems areas, (iii) to propose solutions to some of them, and (iv) outline directions of research for solving others.

The term "execution analysis" covers many important areas, such as debugging, control flow, data flow, performance measurement and storage reference pattern analysis. The main contribution of the thesis is the development of a framework which facilitates analysis tasks in all of these areas. A prototype of this framework, called DAME (Dynamic Analysis and Modelling Environment) has been implemented on the PDP-10 to study the behaviour of PDP-11 programs. Its most novel aspect is that it permits the user to define an abstract structure over his program at run-time and perform his analysis in terms of the elements of that structure, called "nodes". A node is a segment of code, not necessarily contiguous in space, having a unique entry point and a unique exit point. Every execution of a node is called an "instance" of that node. During each node instance, DAME constructs a list, called the "input-set", of all the inputs used, and upon exit, a list, called the "output-set" of the changes to the system state caused by the node instance. The input-set of a node instance I is defined as the set of pairs <A,B> where A is an address whose contents were read by I before being modified for the first time by I, and B is the value read. Thus the input-set of I represents all the outside information passed to I. The output-set of I consists of pairs <C,D> where C is an address written into by I and D the last value written. The significance of this formulation of input/output sets is that it not only permits backtracking to any arbitrary point in the execution history, but also facilitates the determination of data flow between nodes. This formulation is also very helpful in narrowing down the search for an elusive bug to a particular node instance during debugging. Another significant advantage of this approach is that it gives the user the ability to control the amount of information collected by the system through the judicious definition of nodes. Other systems, which record every store and every branch operation, require much more storage to represent the same length of execution.

In addition to the node mechanism, DAME offers a flexible mechanism, called the "Hook Mechanism", which allows the user to trigger monitoring and analysis actions at a wide variety of points in the PDP-11 instruction cycle and at entry and exit from nodes. By using the node and hook mechanisms and the comprehensive

instruction set of DAME, which includes general-purpose computational instructions as well as instructions specifically designed for monitoring, collecting and searching collected data, the user can in most cases easily formulate DAME routines to perform the analysis he is interested in. In Chapter 4, five example of the application of DAME to data flow analysis, control flow analysis and instruction mix analysis are given.

The primary attribute sought in the design of DAME was flexibility. This goal resulted in a list-oriented design; each PDP-11 core location has a, possibly empty, list of "interesting" objects associated with it, e.g. node descriptors, hook descriptors, empirical data saved there by the analysis system, a list of previous values. Each DAME object can have a secondary attribute list which can contain system-defined or user-defined attribute descriptors and arbitrary information associated with the object. The DAME routines themselves and the DAME symbol table are lists manipulable with the standard list functions.

The price for flexibility is usually loss of efficiency. A particularly stiff price was paid for the flexibility afforded by the design of the PDP-11 simulator at the memory cycle level. The motivation for this choice was the prospect that DAME might be used for analyses involving events at Unibus transfer level. Also, it had been envisioned that simulators for several I/O devices capable of generating NPR commands which could interrupt the CPU after a memory cycle within an instruction, could be attached to the basic simulator. In hindsight, this decision seems ill-advised; or perhaps, ironically, too inflexible. This choice, coupled with the general-purpose scheduling mechanism used for timing events, has caused DAME to spend two-thirds of its time in the simulation scheduler, and there is no way to get around this in the present design. A much better design would have been to provide an option to the user as to whether to simulate at memory cycle level or at instruction level or possibly even at subroutine level. This would permit both detailed memory-cycle-level studies over a short simulated time, and debugging, flow analysis and performance measurement studies which require and order of magnitude longer periods of simulated time. These latter areas make up the bulk of the applications of DAME and hence should have been given more emphasis.

The DAME language has proved unsatisfactory in some areas. Its main disadvantage is the low-level instructions supplied for conventional computing tasks (e.g. arithmetic). These are equivalent to those of a 3-address hardware machine. But, this design was in fact intended to provide a model for a possible hardware implementation and it was felt that a higher-level language can subsequently be implemented to compile into the DAME language.

This task has not been done. Such a language would make DAME easier to use.

The subset of DAME instructions dealing with monitoring, data collection and retrieval have proved quite comprehensive in their coverage and easy to use. While this subset could certainly be enhanced by the implementation of higher-level primitives discussed in Chapter 6, such as the FOREACH statement in LEAP and continuously evaluated expressions, the provided facilities have proved quite useful and also quite easy to transport to a higher-level language. Their transportability to a higher-level language as demonstrated in Chapter 7, and the fact that their design was based on the requirements set forth in Chapter 2, indicate that the specifications in Chapter 2 are indeed independent of the analysis language level.

In the final chapter, Chapter 8, we consider a class of questions whose solution could have a significant impact on the extent to which execution analysis facilities are used by application and system programmers alike. These questions relate to the hardware implementation of the primitives which are most burdensome and cause most of the overhead in software. We did not attempt to solve these problems; our intention was only to pose the right questions and suggest approaches to their solution. A real solution to these questions, due to the major design tasks which still remain, would require a detailed, engineering level design and possibly implementation, testing and trial use.

In summary then, we have shown that execution analysis facilities significantly more powerful and widely applicable than the existing systems for individual types of analyses, such as debugging and performance measurement, can be built using current technology. While the prototype implementation appears too expensive for wide use, a more cost-conscious design and some assistance from hardware can bring the cost down substantially. We hope that the ideas demonstrated in this thesis will shed some light on the problems involved and point the way to some of the solutions.

REFERENCES

[ACM 73] Proceedings of "Workshop on Virtual Computer Systems", ACM SIGARCH-SIGOPS, 1973.

[AS 71] Aschenbrenner, R. A., et al, "The Neurotron Monitor System", Proc. FJCC 39(1971).

[BA 67] Balzer, R. M., "EXDAMS-Extendable Debugging and Monitoring System", Proc. FJCC, 1971.

[BE 66] Bernstein, A. J., "Analysis of Programs for Parallel Processing", IEEE Transactions on Electronic Computers, October, 1966.

[BO 68] Bernstein, W. A., Owens, J. J., "Debugging in a Time-Sharing Environment", Proc. FJCC, 1968.

[BK 70] Bussell, B., and Koster, R. A., "Instrumenting Computer Systems and Their Programs", Proc. FJCC, 1970.

[CO 71] Cocke, J., "On Certain Graph-Theoretic Properties of Programs", IBM Research Report RC 3391, 1971.

[DEC 71] "PDP11/20,15,r20 Processor Handbook", Digital Equipment Corp., Maynard, Mass., 1971.

[DEC 73] "BLISS-11 Programmer's Manual", Digital Equipment Corp., Maynard, Mass., 1973.

[ED 66] Evans, T. G., Darley, D. L., "On-line Debugging Techniques: A Survey", Proc. FJCC, 1966.

[ED 65] Evans, T. G., and Darley, D. L., "DEBUG-An Extension to Current Online Debugging Techniques", CACM Vol. 8, No. 5.

[ES 67] Estrin, G., et al, "Snuper Computer-A Computer in Instrumentation Automaton", Proc. SJCC, 1967.

[FE 73] Feustel, E. A., "On the Advantages of Tagged Architecture", IEEE Transactions on Computers, Vol. c-22, No. 7, July 1973.

[FI 70] Fisher, E., "Control Structures for Programming Languages", Ph.D. Thesis, Carnegie-Mellon University, 1970.

[FR 69] Feldman, J. A., and P. D. Rovner, "An ALGOL-Based Associative Language", CACM 12. Vol. 8, August 1969.

[FSW 73] Fuller, S. H., R. J. Swan and W. A. Wulf, "The Instrumentation of C.mmp, a Multi-(Mini) Processor", Proc. of COMPCON 73, IEEE Computer Society.

[GA 69] Gaines, R. Stockton, "The Debugging of Computer Programs", Institute for Defense Analyses, Working Paper No. 266, August, 1969.

[KN 73] Knuth, Donald E., The Art of Computer Programming, Vol. 3, Addison-Wesles, 1973.

[LA 65] Lampson, B. W., "Interactive Machine Language Programming", Proc. FJCC 1965.

[LA 72] Lang, B., "A New Technique for Data Monitoring", ACM SIGPLAN Notices, Vol. 7, No. 6, June 1972.

[LU 71] Lunde, A., "POOMAS-Poor Man's Simula", Unpublished user manual for the POOMAS simulation package available at CMU Computer Science Dept.

[ME 67] Martin, David F. and Estrin, Gerald, "Experiments on Models of Computations and Systems", IEEE Transactions on Electronic Computers, February, 1967.

[MCN 68] McNeley, J. L., "Compound Declarations", in Sumulation Programming Languages, ed. J. N. Buxton, North-Holland, 1968.

[MI 70] Mitchell, J. G., "The Design and Construction of Flexible and Efficient Interactive Programming Systems", Ph.D. Thesis, Carnegie-Mellon University, June 1970.

[RU 71] Rustin, R., "Debugging Techniques for Large Systems", Courant Computer Science Symposium 1, Courant Institute of Mathematical Sciences, New York University, Prentice-Hall, 1971.

[SS 72] Saal, H. J. and L. J. Shustek, "Microprogrammed Implementation of Computer Measurement Techniques", Proc. 5. Annual ACM/SIGMICRO Workshop on Microprogramming, University of Illinois, Urbana, Illinois.

[ST 65] Stockham, T. G., "Some Methods of Graphical Debugging", Proc. IBM Scientific Computing Symposium on Man-Machine Communication, May, 1965.

[WI 67] Wilde, D. U., "Program Analysis Digital Computer", Ph.D. Thesis, MIT, 1967.

[WU 71] Wulf, W. et al., "Bliss Reference Manual", CMU Computer Science Department Research Report, January, 1971.

[WU 72] Wulf, W., "C.mmp: A Multi-Mini-Processor", Computer Science Research Review 1971-72, Department of Computer Science, Carnegie-Mellon University, 1972.

[ZE 71] Zelkowitz, M., "Interrupt Driven Programming" CACM, June 1971 Vol. 14, No. 6.

APPENDIX A

CONTENTS

Introduction to D A M E

DAME (Dynamic Analysis and Modelling Environment) is an environment for running PDP-11/20 programs on the PDP-10 and analyzing their execution. It contains a fairly rich instruction set containing the facilities of a low-level programming language and a set of facilities for controlling the execution on the PDP-11 and the dynamic collection and searching of data. (We shall refer to DAME instructions also as DAME commands). Any DAME command can be executed immediately or in a DAME routine. A DAME routine can either be defined on-line by using the "Define Monitor Routine" (DMR) command or it can be prepared ahead of time in an SOS file with the extension .DAM and subsequently loaded with the "Load Monitor Routine" (LMR) command. The latter mode of operation is highly recommended since SOS has much better editing facilities than DAME and one quickly gets tired of entering the same commands repeatedly. LMR commands can be nested in the sense that any executed routine can load and execute other routines, achieving a hierarchical loading effect. This is a very convenient mode of operation.

PDP-11 programs are loaded from a binary (.BIN) file using the LOAD command. They are executed by using the RUN or GO commands.

The Hook Mechanism

The principal mechanism by which the user causes DAME to take some action while his program is running, is the Hook Mechanism.

There are two classes of hooks: general hooks and addressed hooks. Within each class there are several types. The type and class of each hook is indicated by a mnemonic character constant in the Hook command. General hooks are those in which a user-specified monitor routine will be executed at:

1- Every fetch operation (hook-type 'GF) or,

2- Every store operation (type 'GS) or,

3- Every instruction fetch (type 'IF) or,

4- Every instruction completion (type 'IC) or,

5- Every node entry (type 'NE) or,

6- Every node exit (type 'NX).

(Nodes are explained later.)

Addressed hooks are those in which the user-specified monitor routine will be executed only if the specified type of operation is performed on an address in a given range. The types of operations are:

7- Every fetch from an address range (type 'AF),

8- Every store into an address range (type 'AS),

9- Every instruction fetch from an address range (type 'AIF),

10- The completion of every instruction fetched from an address range (type 'AIC).

The user determines what actions he would like taken at one or more of the above points. He then prepares a monitor routine (by a DMR command or by loading from a .DAM file with a LMR command) and issues a Hook command giving as parameters: the type of hook, the routine name, if an addressed hook then the address range, and a name for the hook (consisting of a character string up to 5 characters preceded by a single quote mark) with which he can refer to the hook later on. He can place as many of any type of hook as he wants. The routines which are thus referenced in a hook specification must be defined prior to the first activation of the hook. In practice, all monitor routines are usually defined prior to the initiation of the execution of the PDP-11 program.

The Node Mechanism

A second important mechanism by which the user collects information about the behaviour of his program, is the so-colled "Node Mechanism". The Node Mechanism reflects a certain view, that held by DAME, of the notion of what "the execution of a program" means. It contains facilities for extracting information in compliance with that view, while the PDP-11 program runs. The collected information makes it possible to reconstruct any previous state of the -11, as well as to answer questions about data flow and control flow history without restoring past states.

In DAME's view of the world, interesting parts of programs are identified and divided into nodes by the user. (A default mode is also provided. See NTR command.) Nodes can be as small as a single instruction or as large as the entire program.

Nodes are defined thru the NODE command, by specifying their entry and exit points. Nodes may be nested but no two nodes may have the same entry or the same exit point; nor may nodes overlap partially. Normally, the last instruction of a node is a branch or subroutine call instruction and the first instruction is the target of a branch instruction or a subroutine entry point. Control need not physically stay within the starting and ending addresses of a node; the entire path followed by the program between the entry and the exit from the node will be considered a part of that "node instance". A "node instance" (NI) is a particular execution of a node. Associated with the concept of an NI are the concepts of the NI's "input-set" and the "output-set". The "input-set" consists of pairs (aj,bj) where aj is an address from which the associated NI has fetched something before writing into it for the first time, and bj is the value fetched from aj for the first time during the NI. Thus, the input-set represents all the "external information" used by the NI. The output-set consists of all the addresses written by a node-instance and the contents of those addresses upon exit from the NI. Thus, it represents all the information passed to the rest of the world by the NI.

For each node instance, the system creates a four-word entry in a table, NODETRACE. The format of the entry is:

<node starting address>
<instruction count at entry>
<input-set ptr,,output-set ptr>
<no. of instructions in NI>

In addition, associated with each node is a node-object, which contains pointers to lists of pointers to the input- and output-sets of every instance of that node. The I/O sets can be displayed easily by the TOBJ(<obj. address>) command by supplying the address of the desired I/O set list from the node-object. These lists can also be manipulated in monitor routines.

Finally, all node-objects and input/output sets are accessible, as most other objects in the system are, thru a set of master list pointers, MNODESC, MINPUTSETSC and MOUTPUTSETSC. These lists, called "subclass masters", contain a pointer to every object of their respective subclasses.

A set of commands intended to facilitate the searching of this execution history information is provided (See "Commands for Searching Execution History").

Data Elements of DAME

DAME has access to three address spaces, each of which is handled in a similar, but not identical, manner. These are:

1- PDP-11 core, general and device registers,

2- Global PDP-10 symbols declared in the simulator and in the rest of DAME code,

3- Monitor Machine objects (MMO) created by the user during the session or pre-defined for the user by DAME during initialization.

A list of the useful elements of type 2 and pre-defined objects of type 3 are found in the back of this document. Symbols of type 1 are identical to the corresponding, standard PDP-11 assembly language symbols as defined in [DEC 71].

Procedure for Getting Started with DAME

To run DAME, enter the following command to the PDP-10 Monitor:

```
.RUN DAME C410BA07
```

It will respond with:

```
DAME11/10...
**
```

and unlock the keyboard. You are now in DAME command mode, indicated by the double-asterisk prompt signal.

(Notation: A BNF-like notation is used to describe the syntax of DAME instructions. "/" indicates disjunction, and "<" and ">" delimit non-terminal symbols. Brackets "[" and "]" delimit optional operands.)

Instruction Format

<MM instruction> → <Type-1 instruction> / <Type-2 instruction>

<Type-1 instruction> → <operator>(<operand list>)

<Type-2 instruction> → <operator>(<operand list> <action>)

<operand list> → <operand>/<operand list> <operand>

<operand> → <octal integer> / @<octal integer> / <short char. string>
<global -10 symbol> / <MMO name>

<action> → <MM routine name> / <compound instruction>

<short char. string> → '<up to 5 characters>

<compound instruction> → (<MM instruction list>)

<MM instruction list> → <MM instruction> / <MM instruction list>
<MM instruction>

As can be seen, some monitor instructions take simple operand lists while others (in particular, IF, INCR, WHL, HOOK and ALONG instructions) can optionally take a compound-instruction (the analogue of a compound statement or compound expression in block-oriented languages) as the last operand. All operands of an MM instruction must be defined prior to the execution of that instruction. MMO's which are not pre-defined by the system, are defined by the CREATE instruction (except for monitor routines, hooks and value-trace objects, as described later.) The form @<octal integer> refers to the contents of -11 core location <octal integer> when the instruction is executed.

MONITOR MACHINE INSTRUCTION SET

****
****

Commands for Creating Monitor Routines

****
****

"Load Monitor Routine" Command

Syntax: LMR ('<filename> <routine spec.>)
<routine spec.> ::= '<routine name>/'*

Effect: There must be an SOS file named <filename>.BIN, where <filename> has at most five characters. The file must contain Monitor routines in the following format:

```
<routine name>(<MM instruction> <MM instruction>
                                .....
                                .....)
<routine name>(<MM instruction> .......
                                .....
                                .....)
```

i.e. each routine must start on a new SOS line.

Standard SOS line numbering is assumed. If '* is specified, all the routines in the file are loaded and defined as MMO's. Otherwise, if the specified routine is found in the file, it is loaded and defined, else an error message is typed.

****

"Define Monitor Routine" Command

Syntax: DMR ('<routine name>)

Effect: <routine name> must be at most 5 characters. The command puts the user in the DAME edit mode, which is indicated by the prompt characters '--' for the first line of a routine being defined. If the routine is to extend into more lines, terminate each non-terminal line with an altmode and carriage-return; DAME will prompt with '---' for each non-terminal line after the first line. Terminate last line with only a carriage return.

****
****

PDP-11 Flow Control Commands

****
****

"Run" Command

Syntax: RUN([<starting address>[<halt count>]])

Effect: If <starting address> is specified, it is inserted in the PDP-11 PC. If <halt count> is specified, it is inserted in the global variable HALTCOUNT, the value of which is initialized to -1 when the system is started up. The CPU is then given control, starting with an instruction fetch from the current value of PC. HALTCOUNT is decremented by 1 after the completion of every instruction. When it reaches zero, execution is stopped and command mode is entered.

****

"Go" Command

Syntax: GO([<halt count>])

Effect: If <halt count> is specified, it is inserted in HALTCOUNT. Execution is resumed from its current state.

****

"Stop" Command

Syntax: STOP( )

Effect: CPU is stopped and command mode is entered.

****

"Stop Comditional" Command

Syntax: STOPC(<id>)

Effect: If the value of <id> is odd, then same as STOP ( ), otherwise no effect.

****

"Node" Command

Syntax: NODE('<node name> <lower bound> <upper bound>)

Effect: Defines a node-object with name <node name> and whose scope is <lower bound> to <upper bound>. See the format of objects of nodesubclass and also NODETRACE table for the format of node-instances (p. 41 and 49).

****

"Node Trace" Command

Syntax: NTR()

Effect: This command causes the system to assume the default mode for node definition. The first executed instruction starts the first node and first node instance. Thereafter, every conditional branch and every deviation from sequential flow causes the termination of the current node instance, and the following instruction (i.e. the target of the transfer) constitutes entry into a new node instance. The current node instance is also terminated when a preciously-established end of a node instance is encountered even if control flow remains sequential.

****

"Along" Command

Syntax: ALONG(N0 N1...Nk R)

Effect: Ni's must be the names or starting addresses of nodes and R a compound-instruction or the name of a monitor routine. Whenever the execution follows path N0,N1,...,Nk, R is executed whenever this ALONG command is encountered. More precisely, let L0,L1,...,Lt be the sequence (in reverse chronological order) of nodes executed so far, with L0= the current node. Then R will be executed if and only if for some j, $0 \leq j \leq k$ for all i=0,...,j, Ni=L(j-i); i.e. if some (j+1)-element initial segment N0,N1,...,Nj in the specified path is identical to L(j),L(j-1),...,L0, the last j+1 node instances executed.

****

"Restore to Node Instance" Command

Syntax: REST(<index>)
<index> → <octal integer> / <obj. name>

Effect: The PDP-11 environment which existed where the node instance specified by <index> was entered is restored, including the NODETRACE table and the instruction count ICOUNT. However, simulation time is not restored.

****

"Replay Node Instance" Command

Syntax: RPLAY(['T] <starting-index> [<ending-index>])

Effect: The input-sets of the node instances back through the instance of index <starting index> and a replay is made of the node-instances specified by <starting-index> thru <ending-index>. (A node instance has index i if it is the ith node instance entered since the first node was defined. The indices of node instances can be determined via the Find Node Instance (FNI) command.) At the end of the replay, the PDP-11 state which existed when the RPLAY command was issued is restored, including the NODETRACE table, instruction count and simulation time. If 'T is specified, the instructions are traced on the TTY as they are executed.

****
****

Monitor Routine Flow Control Commands

****
****

"If" Command

Syntax: IF(<opd1> '<rel> <opd2> <then-action> [<else-action>])

<then-action> → <action>

<else-action> → <action>

<action> → <routine name> / <compound instruction>

<compound-instruction> → (<MM instruction list>)

<MM instruction list> → <MM instruction>
/ <MM instruction list> <MM instruction>

<rel> → EQ/NEQ/GE/GT/LE/LT

Effect: If the specified relation holds then the action <then-action> is executed. Otherwise, if an <else-action> has been specified, it is executed.

****

"While" Command

Syntax: WHL(<opd> <action>)

Effect: The action <action> is executed while the value of <opd> is odd. <action> is defined as above.

****

"Incr" Command

Syntax: INCR(<var> <from-opd> <to-opd> <step-opd> <action>)

Effect: As the value of <var> is incremented from <from-opd> to at most <to-opd> in steps of <step-opd>, <action> is executed at each step. If <from-opd> is initially smaller than <to-opd>, <action> is not executed at all. <action> is defined as above.

****

"Execute" Command

Syntax: EX(<routine>)

Effect: The monitor routine <routine> is executed. This command, together with the PUSH, POP and RET commands described below, constitute a subroutine facility with call-by-value parameters.

****

"Push" Command

Syntax: PUSH(<value>)
<value> → <octal integer> / '<char. const. up to 5 chars.>
/ <obj. name>

Effect: The provided literal or the contents of word 0 of <obj. name> are pushed on a (implied) stack from where they can be retrieved by a POP command.

****

"Pop" Command

Syntax: POP(<obj. id>)

Effect: The last element pushed onto the stack is popped into word 0 of <obj. id>.

****

"Return" Command

Syntax: RET(<level count>)

Effect: Causes an exit from the last <level count> number of monitor routines and compound-instruction levels; the level count for current level being zero. (Note that, in fact, RET(0) is a useless case since it means that the MM instructions following the RET(0) in the same level, will never be executed. The effect of that level would remain unchanged if the RET(0) and all the following instructions in the same level were removed.)

****
****

"Type-Out" Commands

****
****

"Type Object" Command

Syntax: TOBJ(<obj. name or address>)

Effect: Types the contents of the object whose name or -10 address is given, at the terminal in a format appropriate to the class of the object.

List-objects are typed between a pair of brackets, []. Each element of the list is also typed according to these same rules, recursively.

Representative-objects are indicated by a → followed by a recursive type-out of the object they represent.

Numeric-variable objects are typed, for an object named ABC, as 'ABC: ' followed by the contents of ABC where each word is typed in PDP-10 numeric half-word format and words are seperated by slashes. The last word is followed by two spaces.

Character-variables are typed in the same format as numeric-variable objects, except that each user word is interpreted as a left-justified character string and typed out as such.

Numeric-constant and character-constant objects are typed in a format similar to those of the corresponding variables except that no name is typed. Long-character-constant objects are typed without the slashes between user words.

(There are two classes of objects which are not normally used by the user. These are included here only for completeness. Id-objects, which represent names in a monitor instruction, are typed between <...>. Non-homogenous objects are typed, for an n-word object by interpreting user word i as an object class and typing out user word i+1 according to that class followed by a colon, i=0,2,...,n-2. These are used in the Symbol Table to represent entries.)

For an object whose class is something other than one of the above, an error message is typed indicating the class of the object. (For a list of object classes, see Create Object Command.)

Note that TOBJ command must be used to type only MM objects.

Every completed type-out is followed by a carriage-return, line-feed.

****

"Type-Indirect Object" Command

Syntax: TIOBJ(<pointer>)

Effect: Performs "Type Object" Command on the object pointed by <pointer>. This command is especially useful for typing out objects pointed by global PDP-10 symbols, by giving the -10 symbol as the <pointer>. See the list of global variables at the end of this appendix.

****

"Type -10 Symbol" Command

Syntax: TY10(<global var. name or address>)

Effect: The contents of the specified global variable or the -10 address is typed out in octal half-word format, followed by two spaces.

****

"Type Contents of -11 Addresses" Command

Syntax: T(<starting address> [<ending address>])

Effect: Types out the contents of -11 core from <starting address> to <ending address>. Either term may be a constant or an object whose word 0 contains the address. If the latter is omitted, it is taken to be equal to the former. For each core word, the type-out has the form:

<MMO list ptr>,<I/M bits>,<-11 word>.

The first field is the 18-bit -10 address of the list of MMO's associated with that -11 location, e.g. hooks, value-traces, node-objects etc. These may be examined by entering TOBJ(<MMO list ptr>). See "Object Subclasses" for the format of each such object.

<I/M bits> are used in the determination of Input-Output sets, and are not of direct interest to the user.

Each word is followed by two spaces. Words are written eight to a line.

****

"Type Immediate" Command

Syntax: TI(<literal>)
<literal> → <non-neg. octal integer> /
'<char. string up to 4 chars.>

Effect: Types out the supplied literal.

****

"Type Node Instances" Command

Syntax: TNI([<starting index>] <count>)

Effect: <count> number of node instances starting with <starting index> are typed on the TTY (moving forward in time if <count> is positive, otherwise moving backward in time). If <starting-index> is omitted, it is taken to be the setting of the node instance pointer NIP. The format of each typed instance is (typed on one line):

<index> <node address> <flags>
<input-set address> <output-set address>
<no. of instructions in the node instance>

****

"Type Node Objects" Command

Syntax: TNO(<a1> <a2>... <an>)

Effect: Types the node objects associated with PDP-11 addresses a1,...,an.

****
****

"Insert" Commands

****
****

"Insert in -11 Address" Command

Syntax: I(<address> <value>)

Effect: Each operand may be a constant or an object name. In the latter case, the contents of word 0 of the object is used as the -11 address or the value. If the value is less than 177777, the control bits (bits 16-35) of the core are unaffected and the value is placed in the -11 word. Otherwise the full -10 word is replaced by the value.

****

"Zero -11 Addresses" Command

Syntax: Z(<starting address> <ending address>)

Effect: Either operand may be a constant or an object name. The -11 words between the specified objects are set to zero.

****

"Insert in Object" Command

Syntax: IOBJ(<obj. name> <N> <value>)

Effect: The <value> is inserted in word <N> of the object <obj. name>. Either <N> or <value> may be a constant or an object name. <value> may be an octal constant or a character constant of at most 5 characters preceded by the single quote If <value> is the name of an object whose subclass is #13 (ADDR11SUBCLASS), its contents are taken to be an -11 core address, and the contents of that address are used as the value.

****

"Insert Halfword" Command

Syntax: IHW(<obj. id> <start-address> [<n>])

Effect: The nth halfword in the -10, counting left to right, starting with the left-halfword of <start-address>, is inserted in right half of <obj. id>.

****
****

Commands to Create and Delete Objects

****
****

"Create Object" Command

Syntax: CR('<obj. name> [<class> <subclass> <size>])

Effect: Creates and object according to these specifications. If only the first operand is specified, the default values for the other 3 operands are used. These are #100 (numeric constant class), 0 (free subclass) and 1 (1 user word). All specified operands must be constants. <obj. name> must have at most 5 characters.

The object classes which the user may use are:

100. numeric class
300: character class (up to 5 characters)
700: long-character-string class

The object subclasses which the user may use are:

0 : free subclass (i.e. uninterpreted)
13 : PDP-11 address subclass (whenever the object is encountered, the contents of the PDP-11 word pointed by it are taken)
14 : PDP-10 address subclass (whenever the object is encountered, the contents of the PDP-10 word pointed by it are taken)

These classes and subclasses are that subset of all the pre-defined classes and subclasses which should be visible to the user. There are many others which are used by DAME and POOMAS functions. The user may create objects with classes and subclasses other than those pre-defined. In such objects the classes assigned should be between octal 1000 and 77770 and subclasses between octal 70 and 77770 in order to avoid conflicts with the pre-defined ones. Objects with such user-defined classes may not be typed out with the TOBJ command.

****

"Delete Object" Command

Syntax: DEL(<obj. name or address>)

Effect: Deletes the specified object and returns its space to the free-space list.

****
****

Hook Manipulation Commands

****
****

"Hook" Command

Syntax: HOOK(<hook specification>)

<hook specification> → <general hook spec.> /
<addressed hook spec.>

<general hook spec.> → '<gen. hook code> <action>
<hook name>

<gen. hook code> → GF/GS/IF/IC/NE/NX

<addressed hook spec.> → '<addr. hook code>
<action>
<lower bound>
<upper bound>
'<hook name>

<addr. hook code> → AIF/AIC/AF/AS

<hook name> → <char. string up to 5 chars.>

<lower bound> → <octal integer> / <regname>

<upper bound> → <octal integer not smaller than
<lower bound>>

Effect: The HOOK command is the principal means by which the user executes monitor routines during the execution of his program. General hooks, i.e. those with codes 'GF,'GS,'IF,'IC, 'OF,'OS,'NE or 'NX cause the execution of the specified monitor routine at every: fetch, store, instruction fetch, instruction completion, node entry or node exit respectively.

Addressed hooks, i.e. those with codes 'AF,'AS,'AIF or 'AIC, cause the execution of the specified monitor routine whenever a fetch, a store, an instruction fetch or the completion of an instruction occurs from a location within the specified bounds. If register names are used, the following additional rule must be observed: for general registers the bounds must stay within R0 to R7, and other registers, namely TKB, TKS, TPB, TPS and PS, must be specified individually, by giving the same name for both the lower and upper bounds.

****

"Disable Hook" Command

Syntax: DISAB(<hook-obj. name or address>)

Effect: Causes any future activations of the hook to be a no-op.

****

"Enable Hook" Command

Syntax: ENAB(<hook-obj. name or address>)

Effect: Causes the monitor routines associated with the hook to be executed whenever the hook is activated.

****
****

Commands for Searching PDP-11 Execution History

****
****

"Find Input-Set" Command

Syntax: FISET(<obj. id> <node-spec> <search-spec>
[<direction> [<starting index>]])

<node-spec> → '* / <node-id>

<search-spec> → <routine name>
/ <compound-instruction>

<direction> → 'F/'B

<starting index> → <positive octal integer> /
<obj. name>

Effect: A search is made over the input-sets of past node instances until one satisfying <search-spec> is found. If such a node instance is found, the address of its input-set is inserted in <obj. id> and the node-instance pointer NIP (which is a PDP-10 global variable) is set to the index of the node instance; otherwise a 36-bit -1 is inserted in <obj. id> and NIP is unaffected. If a <node-id> is provided in <node-spec>, only the instances of that node are searched; if '* is specified all input sets are searched. If a <direction> is provided, search takes place in that direction ('F for "forward", 'B for "backward"); otherwise search takes place backward. If a <starting-index> is provided, search starts from that index, otherwise it starts from the most recent node instance. (Note that if NIP is specified as <starting-index>, it will start from the current setting of NIP.)

The procedure for the application of the predicate, i.e. <search spec>, is as follows: The system pushes the address of the input-set to be tried on the stack. Hence, the routine or compound instruction supplied in <search-spec>, referred to as the predicate hereafter, must obtain that address by a POP(A) instruction, where A is some input-set name. Then, the contents of an address Q in the input-set can be extracted by the "find value" instruction FVAL(B A Q) which will insert in the object B either the (16 bit) contents of -11 address Q in the set pointed by A if Q is in fact in that set, else -1. The predicate must

obtain the contents of all addresses in this manner and perform normal arithmetic or comparison operations on them, which constitutes the body of the predicate. Then finally, if the desired conditions are met (i.e. the predicate is satisfied), a PUSH(1) otherwise a PUSH(0) must be performed. Upon exit from the predicate, the system will pop the stack. If the popped value is 1, the index of the node instance just searched will be inserted in <obj. id> and the instruction will be terminated. Otherwise, if the end of the node trace history has been reached, <obj. id> will be set to -1; else the address of the next input-set to be searched will be pushed on the stack and the cycle repreated again.

Example: Suppose we wish to find the most recent input-set of an instance of node N where the contents of address 1000 is greater than the contents of address 2000. Provided the objects A, B, N and Y have been previously created and the node N previously defined by a node or NTR instruction, the following instruction should do this:

```
FISET(B N ( POP(A)
            FVAL(X A 1000)
            FVAL(Y A 2000)
            IF(X 'GT Y (PUSH(1)) (PUSH(0)) ) ))
```

This instruction will insert in B either the address of the input-set of the most recent instance of N in which @1000>@2000 or the value -1 if no such input-set can be found.

****

"Find Output-Set" Command

Syntax: FOSET(<obj. id> <node-spec> <search-spec> [<direction> [<starting index>]])

Effect: The same as FISET except that output-sets are searched rather than input-sets.

****

"Find Value" Command

Syntax: FVAL(<obj. id> <address of I/O set> <-11 address>)

Effect: If the specified -11 address appears in the specified I/O set, then the contents of the address in that set, otherwise -1, is inserted in <obj. id>.

****

"Find Node Instance" Command

Syntax : FNI(<obj. id> <node-spec> <instance-count>
[<starting-index> [<direction>]])

<node-spec> : <node-id> / CURNODE

<instance-count> : <octal integer> / <obj. name>

Effect: An attempt is made to find the nth instance of the node specified by <node-spec> where n= <octal integer> if one is supplied, otherwise contents of word 0 of <obj. name>. CURNODE means the current node. If a <starting-index> is specified, the search starts from there, otherwise from the current node instance. NIP is a valid parameter for <starting-index>. If a <direction> is specified, the search proceeds in that direction; otherwise it proceeds in the backward direction. If the desired node instance is found, its index is inserted into <obj. id> and into NIP. Otherwise, -1 is inserted into <obj. id> and NIP is unaffected.

****

"Find Node Object" Command

Syntax: FNO(<obj. id> <-11 address>)

Effect: If <-11 address> is the starting address of a node, the address of the node object, otherwise -1, is inserted in <obj. id>.

****
****

"Value-trace" Commands

****
****

"Initialize Value-trace" Command

Syntax: IVT(<-11 addr.> <number> '<obj. name>)

Effect: Creates a value-trace object with name <obj. name> with enough room for <number> previous values and puts the object in the MMO list of the specified -11 address or register. Note: This command does not initiate the collection of values. It merely creates and object to hold those values. The collection of values is initiated by the VTH command.

****

"Value-trace Hook" Command

Syntax: VTH(<-11 addr. or reg. name>)

Effect: Causes the monitoring of values stored into the specified core location or register and maintains a circular buffer of the last k values, unique or non-unique, stored there by the PDP-11, where k is the <number> specified in the preceding IVT command for the same address. All accesses by the -11 to write into the specified cell (including auto register incrementation, decrementation, turning on/off bits in the condition code or the device registers) are considered store operations and cause a new entry in the value-trace.

****
****

Disk I/O Commands

****
****

"Write Disk" Command

Syntax: WDSK(<obj. id>)

Effect: Will write (in PDP-10 dump mode) on disk file USER.DAM the contents of the object whose name or address is given in <obj. id>. If the file does not exist, it will be created; otherwise its old contents will be destroyed.

****

"Write-Indirect Disk" Command

Syntax: WIDSK(<address>)

Effect: Will perform WDSK(<obj. id>) where <address> contains a pointer to <obj. id>. This command is particularly useful for writing out objects pointed by PDP-10 symbols.

****

"Read Disk" Command

Syntax: RDSK(<obj. id>)

Effect: Will read a 36-bit word from the binary file USER.DAM (which had better exist!) into the object <obj. id>.

****
****

Miscellaneous Commands

****
****

"Load PDP-11 Program" Command

Syntax: LOAD('<file name> <starting address>)

Effect: The file must be in the absolute unpacked output format of the PAL-11 assembler or MACX11 with /I/A switches, must have extension .BIN and the <file name> must be at most 5 characters. <starting address> must be an even octal integer between 0 and 157776 - X, where X is the length of the program in bytes.

****

Generalized "Unary Operation with Assignment" Command

Syntax: UA('<operation> <target> <opd>)

<operation> → SUC / PRED / SAL / SIZE / ADDR / NOT

<target> → <obj. id>

<opd> → <obj. id>

Effect: The specified unary operation is performed on <opd> and the result is inserted in <target>. 'SUC and 'PRED are the successor and predecessor functions, respectively. 'SIZE returns the number of user words in <opd>, 'ADDR the address of <opd>. 'SAL the address of the secondary-attribute-list (SAL) of <opd> and 'NOT the logical complement of the contents of the first user word of <opd>.

****

Generalized "Binary Operation with Assignment" Command

Syntax: BA('<operation> <target> <opd1> <opd2>)

<operation> → + / - / * / <slash> / AND / OR / XOR / ! / #

where <slash> is the integer division sign "/".

Effect: The command performs <target> ← <opd1> <operation> <opd2>. A#B is the Bth previous value of -11 core location A. ! is the vector index operation and returns the contents of word <opd2> of <opd1>. No bounds check is made. Any or all of <target>, <opd1> and <opd2> may be octal constants, MMO names or global variables. A constant for the <target> is interpreted as a -10 address; for the others as a literal. All the arithmetic and logical operations are defined the same as in BLISS -10. In the case of # operation, a value-trace hook for at least <opd2> previous values must have been placed on <opd1> and a VTH command must have been issued (See IVT and VTH Commands). If <opd2> exceeds the number of values declared to be kept in the last IVT command for the location <opd1>, and error message will be typed out and no assignment will be made. If <opd2> number of values have not yet been stored into the specified location, the half word #777777 will be returned.

****

"Execute External" Command

Syntax: XX(<PDP-10 routine name> [<param. list>])

<param. list> → <param> / <param. list> <param>

<param> → <literal> / <identifier>

Effect: Calls the specified routine with the given parameters. Caution: If identifiers are given as parameters, their addresses are passed. If you wish the contents of the identifier passed, include an additional parameter, namely the literals'. (single quote followed by a dot and a space) before each such parameter. This convention applies only to this and to the EVAL command below.

****

"Evaluate" Command

Syntax: EVAL(<target> <PDP-10 routine name> [<param. list>])

Effect: The -10 routine is called in the same manner as in Execute External. The only difference is that the value returned by the routine is stored in <target>. The value returned by a routine is assumed to be in register 3, following BLISS/10 convention.

****

"Time" Command

Syntax: TIME(<obj. id> '<scale> '<type>)

<scale> → MICS / MILS
(for microseconds or milliseconds respectively)

<type> → FIX / FLOAT

Effect: Puts in word 0 of <obj. id> the current value of the simulation clock according to the given specifications (i.e. in microseconds or milliseconds and in fixed or floating point).

****

"Plot" Command

Syntax: PLOT(<space count> '<char>)

<space count> → <literal> / <identifier>
<space count>

Effect: Types carriage-return, line-feed, "!", followed by <space count> spaces and the character <char>.

A LIST OF USEFUL GLOBAL PDP-10 SYMBOLS AND THEIR CONTENTS

SYMBOL	CONTENTS
(For addressed fetch hooks)	
AFHDATA	The data just fetched
AFHADDR	The address of the fetch
(For addressed store hooks)	
ASHDATA	The data to be stored
ASHADDR	The address of the store
DATA	Contents of Unibus Data lines
ADDR	Contents of Unibus Address lines
CONT	Contents of Unibus Control lines
OLDPC	Last value of the Program Counter (R7)
OPN	A unique integer between 0 and octal 111 representing the current opcode (See next table)
OPC	The assembly language mnemonic for the current opcode
DSTREG, DSTMODE, DSTDATA	The destination register, mode and operand-value, respectively, of the most recent (including the current) single-operand or double-operand instruction
SRCREG, SRCMODE, SRCDATA	The source register, mode and operand-value of the most recent (including the current) double-operand instruction
HALTCOUNT	Number of instructions after which simulator will stop (normally maintained by DAME but may be set by user)
CURNODE	PDP-11 address of the current node
CURNOBJ	PDP-10 address of the node object for the current node
CISP	Pointer to current input-set
COSP	Pointer to current output-set

THE OCTAL VALUE OF OPN FOR EACH OPCODE (OPN=i+j)

i \ j	0	1	2	3	4	5	6	7
0	MOV	MOVB	CMP	CMPB	BIT	BITB	BIC	BICB
10	BIS	BISB	ADD	SUB	CLR	CLRB	COM	COMB
20	INC	INCB	DEC	DECB	NEG	NEGB	ADC	ADCB
30	SBC	SBCB	TST	TSTB	RDR	RDRB	ROL	ROLB
40	ASR	ASRB	ASL	ASLB	JMP	SWAB	No-op	CLC
50	CLV	CLZ	CLN	No-op	SEC	SEV	SEZ	SEN
60	BR	BNE	BEQ	BGE	BLT	BGT	BLE	BPL
70	BMI	BHI	BLOS	BVC	BVS	BCC	BCS	Not used
100	JSR	RTS	HALT	WAIT	RTI	(break point trap)	IOT	RESET
110	EMT	TRAP						

APPENDIX B

Syntax of MDL

module → MODULE name = e ELUDOM

block → BEGIN blockbody END / (blockbody)

compoundexpression → BEGIN expressionsequence END

blockbody → declarations; expressionsequence

declarations → declaration / declaration; declarations

expressionsequence → / e / e; expressionsequence

e → simpleexpression / controlexpression / name: e

simpleexpression → p10 ← e / p10

p10 → p9 / p10 OR p9

p9 → p8 / p9 AND p8

p8 → p7 / NOT p7

p7 → p6 / p6 relation p6

p6 → p5 / - p5 / p6 + p5 / p6 - p5

p5 → p4 / p5 * p4 / p5 / p4 / p5 MOD p4

p4 → p3 / p4 ↑ p3

p3 → decimal / name / name [elist] / e (elist)
/ e() / block / compoundexpression

elist → e / elist, e

relation → EQL / NEQ / LSS / LEQ / GTR / GEQ

controlexpression → conditionalexpression / loopexpression /
choiceexpression / escapeexpression

conditionalexpression → IF e_1 THEN e_2 / IF e_1 THEN e_2 ELSE e_3

```
loopexpression → WHILE e   DO e
                       1       2

loopexpression → INCR name FROM e   TO e   BY e   DO e
                                1       2       3       4

escapeexpression → EXIT level escapevalue /
                   RETURN escapevalue / LEAVE name escapevalue

level → / [e]

escapevalue → / e

choiceexpression → SELECT elist OF NSET nexpressionset TESN

nexpressionset → / ne / ne;  nexpressionset

ne → e:e

declaration → routinedeclaration / allocationdeclaration

allocationdeclaration → allocatetype idlist

allocatetype → GLOBAL / LOCAL / OWN / EXTERNAL / LABEL

idlist → id / idlist , id

id → name / name [dimensionlist]

dimensionlist → decimal / dimensionlist, decimal

routinedeclaration → ROUTINE name (namelist) = e /
                     ROUTINE name = e /
                     EXTERNAL flist

flist → name / flist, name

name → letter / name letter / name digit

letter → A / B / ... / Z

digit → 0 / 1 / ... / 9

decimal → digit / decimal digit
```